KERNEL MACHINES THAT ADAPT TO GPUS FOR EFFECTIVE LARGE BATCH TRAINING

Siyuan Ma [1] **Mikhail Belkin** [1]

ABSTRACT

Modern machine learning models are typically trained using Stochastic Gradient Descent (SGD) on massively parallel computing resources such as GPUs. Increasing mini-batch size is a simple and direct way to utilize the parallel computing capacity. For small batch an increase in batch size results in the proportional reduction in the training time, a phenomenon known as *linear scaling*. However, increasing batch size beyond a certain value leads to no further improvement in training time. In this paper we develop the first analytical framework that extends linear scaling to match the parallel computing capacity of a resource. The framework is designed for a class of classical kernel machines. It automatically modifies a standard kernel machine to output a mathematically equivalent prediction function, yet allowing for extended linear scaling, i.e., higher effective parallelization and faster training time on given hardware.

The resulting algorithms are 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 the kernel parameter is needed, further facilitating the practical use of these methods.

1 INTRODUCTION

Modern machine learning models are trained using Stochastic Gradient Descent (SGD) on parallel computing resources such as GPUs. During training we aim to minimize the (wall clock) training time T_{train} given a computational resource, e.g., a bank of GPUs. Although using larger batch size m improves resource utilization, it does not necessarily lead to a reduction in training time. Indeed, we can decompose the training time $T_{\text{train}}(m)$ into two parts,

$$T_{\text{train}}(m) = N_{\text{epoch}}(m) \times T_{\text{epoch}}(m)$$

where $N_{\text{epoch}}(m)$ is the number of training epochs required for convergence and $T_{\text{epoch}}(m)$ is the wall clock time to train for one epoch. It is easy to see that increasing m always leads to higher resource utilization, thus decreasing $T_{\text{epoch}}(m)$. However, $N_{\text{epoch}}(m)$ may increase with m. In fact, for a general class of convex problems it can be shown (Ma et al., 2017) $N_{\text{epoch}}(m)$ is approximately constant for m no more than a certain critical size m^* and $N_{\text{epoch}}(m) \propto m$ for $m > m^*$. On the other hand $T_{\text{epoch}}(m)$ is at best[1] decreases proportionally to $1/m$. Thus the training time is at least

$$T_{\text{train}}(m) = \begin{cases} 1/m, & \text{for } m \le m^* \\ \text{const}, & \text{for } m > m^* \end{cases}$$

In other words, we obtain linear speedup ("linear scaling") for batch sizes up to m^*, beyond which the training time cannot be improved by further increasing m. Furthermore, an important property of m^* is its near independence from the number of training samples as it is primarily determined by the model and the data distribution.

Similar relationship between the batch size and the training time has been observed empirically in training deep neural networks (Krizhevsky, 2014). A heuristic called the "linear scaling rule" has been widely used in deep learning practice (Goyal et al., 2017; You et al., 2017; Jia et al., 2018). Moreover, in parallel to the convex case analyzed in (Ma et al., 2017), recent work (Golmant et al., 2018; McCandlish et al., 2018) empirically demonstrates that m^* is independent of the data size for deep neural networks.

Many best practices of modern large-scale learning (Goyal et al., 2017; You et al., 2017; Jia et al., 2018) start with estimating m^* by either heuristic rules or experiments. The

[1]Department of Computer Science and Engineering, Ohio State University, Columbus, Ohio, United States. Correspondence to: Siyuan Ma <masi@cse.ohio-state.edu>, Mikhail Belkin <mbelkin@cse.ohio-state.edu>.

Proceedings of the 2^{nd} SysML Conference, Palo Alto, CA, USA, 2019.

[1]Assuming perfect parallel computation.

optimal training time for the model is then capped by the estimated batch size m^* which is fixed given the model architecture and weight, as well as the learning task.

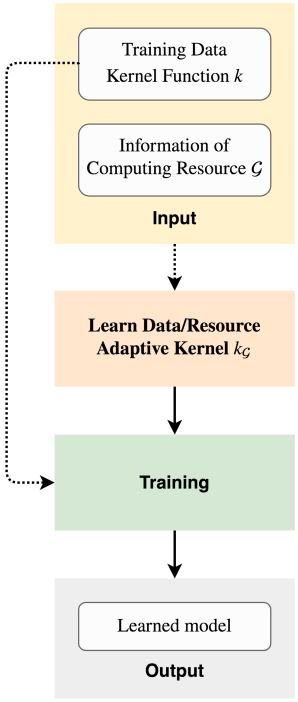
In this work we propose a principled framework (EigenPro 2.0) that increases m^* for a class of models corresponding to classical kernel machines. Our framework modifies a kernel machine to output a mathematically equivalent prediction function, yet allowing for extended linear scaling adaptive to (potentially) arbitrary parallel computational resource. Furthermore, the optimization parameter selection is analytic, making it easy and efficient to use in practice and appropriate for "interactive" exploratory machine learning and automatic model selection. The resulting algorithms show significant speedup for training on GPUs over the state-of-the-art methods and excellent generalization performance.

Kernel machines. 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 small 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 (Tu et al., 2016; Avron et al., 2016) or super-computer nodes (Huang et al., 2014). Very recently, FALKON (Rudi et al., 2017) and EigenPro (Ma & Belkin, 2017) showed strong classification results on large datasets with much lower computational requirements, a few hours on a single GPU.

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 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[2] 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 (Zhang et al., 2016). It has been observed in (Belkin et al., 2018) 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 (Yao et al., 2007) are provided by early stopping, i.e., stopping iterations well 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 (Ma et al., 2017) 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

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

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 (Ma & Belkin, 2017). 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.

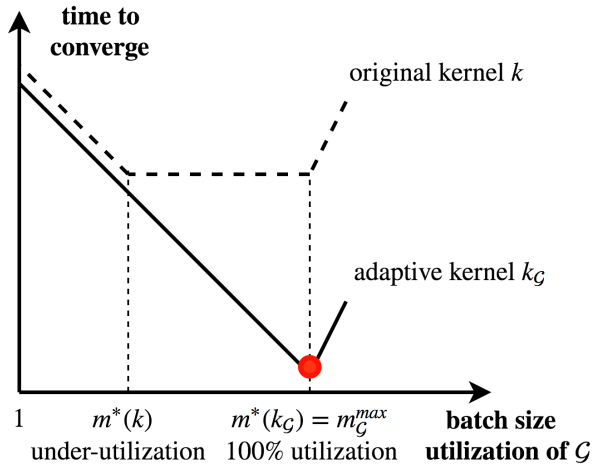


Figure 1: Adaptive and original kernel

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 5.

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 (Ma & Belkin, 2017) significantly reducing the memory requirements and making the computational overhead over the standard SGD negligible.

Comparison to related work. In recent years there has been significant progress on scaling and accelerating kernel methods including (Takác et al., 2013; Huang et al., 2014; Lu et al., 2014; Tu et al., 2016; Avron et al., 2016; May et al., 2017). 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[3]. Two recent methods which allow for high efficiency kernel training with a single CPU or GPU is EigenPro (Ma & Belkin, 2017) (used a as basis for the adaptive kernels in this paper) and FALKON (Rudi et al., 2017). 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 (Krizhevsky, 2014; You et al., 2017; Smith et al., 2017) 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 (Goyal et al., 2017).

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. In Section 4, we present an improved version of EigenPro iteration used by the main algorithm. We then provide comparisons to state-of-the-art kernel methods on several large datasets in Section 5. 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) $\mathscr{H}$ (Aronszajn, 1950) corresponding to a positive definite kernel function

[3] See http://aws.amazon.com/ec2 for details.

$k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$. There is a unique (minimum norm) interpolated solution in $\mathscr{H}$ of the form

$$f^*(\cdot) = \sum_{i=1}^{n} \alpha_i^* k(\boldsymbol{x}_i, \cdot),$$
$$\text{where } (\alpha_1^*, \ldots, \alpha_n^*)^T = K^{-1}(y_1, \ldots, y_n)^T$$

Here K denotes an $n \times n$ kernel matrix, $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. It is easy to check that $\forall_i f^*(x_i) = y_i$.

Remark 2.1 (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

$$f \leftarrow f - \eta \cdot \frac{2}{n}\sum_{i=1}^{n}(f(\boldsymbol{x}_i) - y_i)k(\boldsymbol{x}_i, \cdot) \tag{1}$$

Mini-batch SGD. Instead of calculating the gradient with n training points, each SGD iteration updates the solution f using m subsamples $(\boldsymbol{x}_{t_1}, y_{t_1}), \ldots, (\boldsymbol{x}_{t_m}, y_{t_m})$,

$$f \leftarrow f - \eta \cdot \frac{2}{m}\left\{\sum_{i=1}^{m}(f(\boldsymbol{x}_{t_i}) - y_{t_i})k(\boldsymbol{x}_{t_i}, \cdot)\right\} \tag{2}$$

It is equivalent to randomized coordinate descent (Leventhal & Lewis, 2010) for $K\boldsymbol{\alpha} = \boldsymbol{y}$ on m coordinates of $\boldsymbol{\alpha}$,

$$\alpha_{t_i} \leftarrow \alpha_{t_i} - \eta \cdot \frac{2}{m}\{f(\boldsymbol{x}_{t_i}) - y_{t_i}\} \text{ for } i = 1, \ldots, m \tag{3}$$

Critical mini-batch size as effective parallelism. Theorem 4 in (Ma et al., 2017) shows that for mini-batch iteration (2) with kernel k there is a data-dependent batch size $m^*(k)$ such that

- Convergence per iteration improves linearly with increasing batch size$m$ for $m \le m^*(k)$ (using optimal constant step size).
- 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),

$$m^*(k) = \frac{\beta(K)}{\lambda_1(K)} \text{ where } \beta(K) \triangleq \max_{i=1,\ldots,n} k(\boldsymbol{x}_i, \boldsymbol{x}_i)$$

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 (Ma & Belkin, 2017). 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,

$$\mathscr{P}(f) \triangleq f - \sum_{i=1}^{q}(1 - \frac{\lambda_q}{\lambda_i})\langle e_i, f\rangle_{\mathscr{H}} e_i \tag{4}$$

where $\lambda_1 \ge \cdots \ge \lambda_n$ 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 $\mathscr{H}$,

$$f \leftarrow f - \eta \cdot \mathscr{P}\left\{\frac{2}{m}\sum_{i=1}^{m}(f(\boldsymbol{x}_{t_i}) - y_{t_i})k(\boldsymbol{x}_{t_i}, \cdot)\right\} \tag{5}$$

Remark 2.2 (Data adaptive kernel for fast optimization)**.** EigenPro iteration for target function y and kernel k is equivalent to Richardson iteration / randomized (block) coordinate descent for linear system $K_{\mathscr{P}}\boldsymbol{\alpha} = \boldsymbol{y}_{\mathscr{P}} \triangleq (\mathscr{P}f^*(\boldsymbol{x}_1), \ldots, \mathscr{P}f^*(\boldsymbol{x}_n))^T$. Here $K_{\mathscr{P}}$ is the kernel matrix corresponding to a data-dependent kernel $k_{\mathscr{P}}$. When $n \to \infty$, it has the following expansion according to Mercer's theorem,

$$k_{\mathscr{P}}(\boldsymbol{x}, \boldsymbol{z}) = \sum_{i=1}^{q}\lambda_q e_i(\boldsymbol{x})e_i(\boldsymbol{z}) + \sum_{i=q+1}^{\infty}\lambda_i e_i(\boldsymbol{x})e_i(\boldsymbol{z}) \tag{6}$$

For $n < \infty$, it is a modification of the original kernel k,

$$k_{\mathscr{P}}(\boldsymbol{x}, \boldsymbol{z}) = \mathscr{P}\{k(\boldsymbol{x}, \cdot)\}(\boldsymbol{z})$$
$$\approx k(\boldsymbol{x}, \boldsymbol{z}) - \sum_{i=1}^{q}(\lambda_i - \lambda_q)e_i(\boldsymbol{x})e_i(\boldsymbol{z})$$

Remark 2.3 (Preconditioned linear system / gradient descent)**.** $K_{\mathscr{P}}\boldsymbol{\alpha} = \boldsymbol{y}_{\mathscr{P}}$ is equivalent to the preconditioned linear system $PK\boldsymbol{\alpha} = P\boldsymbol{y}$ where P is a left matrix preconditioner related to $\mathscr{P}$. Accordingly, $\mathscr{P}$ is the operator preconditioner for preconditioned (stochastic) gradient descent (5).

Abstraction for parallel computational resources. To construct a resource adaptive kernel, we consider the following abstraction for given computational resource $\mathcal{G}$,

- $C_{\mathcal{G}}$: Parallel capacity of $\mathcal{G}$, i.e., the number of parallel operations that is required to fully utilize the computing capacity of $\mathcal{G}$.
- $S_{\mathcal{G}}$: 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:

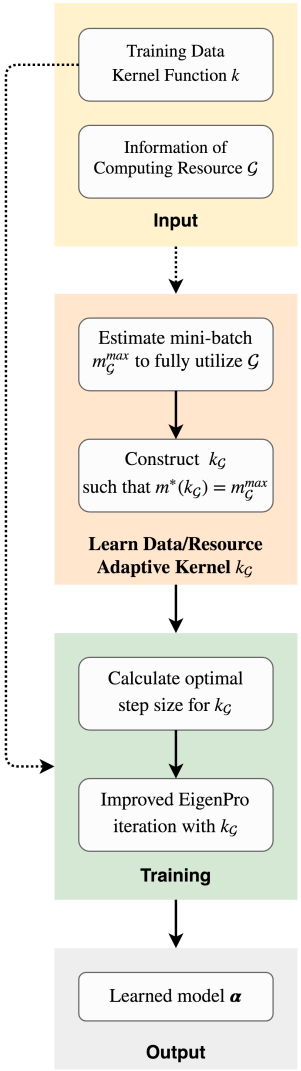
Step 1. Calculate the resource-dependent mini-batch size $m_{\mathcal{G}}^{max}$ to fully utilize resource $\mathcal{G}$.

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 Section 4).

Note that due to properties of EigenPro iteration, training with this adaptive kernel converges to the same solution as the original kernel.

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 (introduced in Section 4) makes computation and memory overhead over the standard SGD negligible (see Table 1). Thus we assume that EigenPro has the same complexity as the standard SGD per iteration.



EigenPro 2.0

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 as in Iteration (2). 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. Additionally we need to store a $m \cdot n$ kernel matrix for the prediction on the mini-batch. In total, we need $(d + l + m) \cdot n$ memory.

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:

- $m_{C_{\mathcal{G}}}$: batch size for fully utilizing parallelism in $\mathcal{G}$ such that $(d + l) \cdot m_{C_{\mathcal{G}}} \cdot n \approx C_{\mathcal{G}}$.
- $m_{S_{\mathcal{G}}}$: 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

$$q \triangleq \max \{i \in \mathbb{N}, \text{ s.t. } m^*(k_{\mathscr{P}_i}) \leq m_{\mathcal{G}}^{max}\} \tag{7}$$

To compute q recall that $m^*(k_{\mathscr{P}_q}) = \frac{\beta(K_{\mathscr{P}_q})}{\lambda_1(K_{\mathscr{P}_q})}$, where $K_{\mathscr{P}_q}$ is the kernel matrix corresponding to the kernel function $k_{\mathscr{P}_q}$. Using the definition of $\mathscr{P}_q$ and β in Section 2, we have

$$\begin{aligned}
\lambda_1(K_{\mathscr{P}_q}) &= \lambda_q(K) \\
\beta(K_{\mathscr{P}_q}) &\approx \max_{i=1,\ldots,n} k_{\mathscr{P}_q}(\boldsymbol{x}_i, \boldsymbol{x}_i) \\
&= \max_{i=1,\ldots,n} \{k(\boldsymbol{x}_i, \boldsymbol{x}_i) - \sum_{j=1}^{q} (\lambda_j - \lambda_q) e_i(\boldsymbol{x}_i)^2\}
\end{aligned}$$

In practice, $\beta(K_{\mathscr{P}_q})$ can be accurately estimated using the maximum of $k_{\mathscr{P}_q}(\boldsymbol{x}, \boldsymbol{x})$ on a small number of subsamples. Similarly, we can estimate $\lambda_q(K)$ on a subsample kernel matrix. Knowing the approximate top eigenvalues of K, allows us to efficiently compute $m^*(k_{\mathscr{P}_p})$ for each p, thus allowing to choose q from (7).

Step 3: Training with adaptive kernel $k_{\mathcal{G}} = k_{\mathscr{P}_q}$. We use the learned kernel $k_{\mathcal{G}}$ with improved EigenPro (Section 4). Its optimization parameters (batch and step size) are calculated as follows:

$$m = m_{\mathcal{G}}^{max}, \eta = \frac{m_{\mathcal{G}}^{max}}{\beta(K_{\mathcal{G}})}$$

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

$$\text{acceleration of } k_{\mathcal{G}} \text{ over } k = \frac{\beta(K)}{\beta(K_{\mathcal{G}})} \cdot \frac{m_{\mathcal{G}}^{max}}{m^*(k)}$$

See the Appendix C 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.

Remark 3.1 (Choice of q)**.** Note that it is not important to select q exactly, according to Eq. 7. In fact, choosing $k_{\mathscr{P}_p}$ for any $p > q$ allows for the same acceleration as $k_{\mathscr{P}_q}$ as long as the mini-batch size is chosen to be $m_{\mathcal{G}}^{max}$ and the step size is chosen accordingly. Thus, we can choose any value $p > q$ for our adaptive kernel $k_{\mathscr{P}_p}$. However, choosing p larger than q incurs an additional computation cost as p eigenvalues and eigenvectors of K need to be approximated accurately. In particular, larger subsample size s (see Section 4 may be needed for approximating eigenvectors.

4 Improved EigenPro Iteration using Nyström Extension

In this section, we present an improvement for the EigenPro iteration originally proposed in (Ma & Belkin, 2017). 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 preconditioner $\mathscr{P}_q$ using Nyström extension.

We start by recalling the EigenPro iteration in RKHS and its preconditioner constructed by the top-q eigensystem λ_i, e_i of the kernel operator $\mathscr{K}$:

$$f \leftarrow f - \eta \cdot \mathscr{P}_q \left\{ \frac{2}{m} \sum_{i=1}^{m} (f(\boldsymbol{x}_{t_i}) - y_{t_i}) k(\boldsymbol{x}_{t_i}, \cdot) \right\}$$

$$\text{where } \mathscr{P}_q(f) = f - \sum_{i=1}^{q} (1 - \frac{\lambda_q}{\lambda_i}) \langle e_i, f \rangle_{\mathscr{H}} e_i$$

The key to construct the above iteration is to obtain an accurate and computationally efficient approximation of λ_i, e_i such that $\mathscr{K} e_i \approx \lambda_i e_i$. The original EigenPro iteration learns an approximate e_i of the form $\sum_{j=1}^{n} w_j k(\boldsymbol{x}_j, \cdot)$. In contrast, our improved version of Eigenpro uses only a small number of subsamples $\boldsymbol{x}_{r_1}, \ldots, \boldsymbol{x}_{r_s}$ to learn an e_i of the form $\sum_{j=1}^{s} w_j k(\boldsymbol{x}_{r_j}, \cdot)$. This compact representation (s versus n) nearly eliminates per-iteration overhead of EigenPro over SGD. Importantly, there is no associated accuracy reduction as this is the same subset used in the original EigenPro to approximate $\mathscr{P}_q$.

Algorithm 1 Improved EigenPro iteration (double coordinate block descent)

Input: Kernel function $k(\boldsymbol{x}, \boldsymbol{z})$, EigenPro parameter q, mini-batch size m, step size η, size of fixed coordinate block s

initialize model parameter $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)^T \leftarrow 0$
subsample s coordinate indices $r_1, \ldots, r_s \in \{1, \ldots, n\}$ for constructing $\mathscr{P}_q$, which form fixed coordinate block $\boldsymbol{\alpha}_r \triangleq (\alpha_{r_1}, \ldots, \alpha_{r_s})^T$
compute top-q eigenvalues $\Sigma \triangleq \text{diag}(\sigma_1, \ldots, \sigma_q)$ and corresponding eigenvectors $V \triangleq (\boldsymbol{e}_1, \ldots, \boldsymbol{e}_q)$ of subsample kernel matrix $K_s = [k(\boldsymbol{x}_{r_i}, \boldsymbol{x}_{r_j})]_{i,j=1}^{s}$

for $t = 1, \ldots$ **do**

1. sample a mini-batch $(\boldsymbol{x}_{t_1}, y_{t_1}), \ldots, (\boldsymbol{x}_{t_m}, y_{t_m})$
2. calculate predictions on the mini-batch

$$f(\boldsymbol{x}_{t_j}) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}_{t_j}) \text{ for } j = 1, \ldots, m$$

3. update **sampled coordinate block** corresponding to the mini-batch $\boldsymbol{\alpha}_t \triangleq (\alpha_{t_1}, \ldots, \alpha_{t_m})$,

$$\boldsymbol{\alpha}_t \leftarrow \boldsymbol{\alpha}_t - \eta \cdot \frac{2}{m} (f(\boldsymbol{x}_{t_1}) - y_{t_1}, \ldots, f(\boldsymbol{x}_{t_m}) - y_{t_m})^T$$

4. evaluate the following feature map $\phi(\cdot)$ on the mini-batch features $\boldsymbol{x}_{t_1}, \ldots, \boldsymbol{x}_{t_m}$:

$$\phi(\boldsymbol{x}) \triangleq (k(\boldsymbol{x}_{r_1}, \boldsymbol{x}), \ldots, k(\boldsymbol{x}_{r_s}, \boldsymbol{x}))^T$$

5. update **fixed coordinate block** $\boldsymbol{\alpha}_r$ to apply $\mathscr{P}_q$,

$$\boldsymbol{\alpha}_r \leftarrow \boldsymbol{\alpha}_r + \eta \cdot \frac{2}{m} \sum_{i=1}^{m} (f(\boldsymbol{x}_{t_i}) - y_{t_i}) \cdot V D V^T \phi(\boldsymbol{x}_{t_i})$$

$$\text{where } D \triangleq (1 - \sigma_q \cdot \Sigma^{-1}) \Sigma^{-1}$$

end for

Next, we show how to approximate λ_i, e_i. We first consider a related linear system for subsamples $\boldsymbol{x}_{r_1}, \ldots, \boldsymbol{x}_{r_s} \in \mathbb{R}^d$: $K_s \boldsymbol{e}_i = \sigma_i \boldsymbol{e}_i$ where $K_s \triangleq [k(\boldsymbol{x}_{r_i}, \boldsymbol{x}_{r_j})]_{i,j=1}^{s}$ is a subsample kernel matrix and $\sigma_i, \boldsymbol{e}_i$ is its eigenvalue/eigenvector. This rank-s linear system is in fact a discretization of $\mathscr{K} e_i = \lambda_i e_i$ in the RKHS.

The two eigensystems, $\sigma_i, \boldsymbol{e}_i$ and λ_i, e_i are connected through *Nyström extension*. Specifically, the Nyström extension of e_i on subsamples $\boldsymbol{x}_{r_1}, \ldots, \boldsymbol{x}_{r_s}$ approximates e_i as follows:

$$e_i(\cdot) \approx \frac{1}{\sigma_i} \sum_{j=1}^{s} e_i(\boldsymbol{x}_{r_j}) k(\boldsymbol{x}_{r_j}, \cdot)$$

Evaluating both side on $\boldsymbol{x}_{r_1}, \ldots, \boldsymbol{x}_{r_s}$, we have

$$\lambda_i \approx \frac{\sigma_i}{s}, e_i(\cdot) \approx \frac{1}{\sqrt{\sigma_i}} \boldsymbol{e}_i^T \phi(\cdot)$$

where $\phi(\cdot) \triangleq (k(\boldsymbol{x}_{r_1}, \cdot), \ldots, k(\boldsymbol{x}_{r_s}, \cdot))^T$ is a kernel feature map. Thus we approximate the top-q eigensystem of $\mathscr{K}$ using the top-q eigensystem of K_s. These (low-rank) approximations further allow us to apply $\mathscr{P}_q$ for efficient EigenPro iteration on mini-batch $(\boldsymbol{x}_{t_1}, y_{t_1}) \ldots, (\boldsymbol{x}_{t_m}, y_{t_m})$,

$$\begin{aligned} f &\leftarrow f - \eta \cdot \frac{2}{m} \sum_{i=1}^{m} (f(\boldsymbol{x}_{t_i}) - y_{t_i}) k(\boldsymbol{x}_{t_i}, \cdot) \\ &+ \eta \cdot \frac{2}{m} \sum_{i=1}^{m} (f(\boldsymbol{x}_{t_i}) - y_{t_i}) \cdot \phi(\boldsymbol{x}_{t_i})^T V D V^T \phi(\cdot) \\ &\text{where } D \triangleq \Sigma^{-1}(1 - \sigma_q \cdot \Sigma^{-1}) \end{aligned} \tag{8}$$

where $\Sigma \triangleq \text{diag}(\sigma_1, \cdots, \sigma_q)$ and $V \triangleq (\boldsymbol{e}_1, \cdots, \boldsymbol{e}_q)$ are top-q eigensystem of K_s.

Recalling that $f = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}_i, \cdot)$, the above iteration can be executed by updating two coordinate blocks of the parameter vector $\boldsymbol{\alpha}$ as in Algorithm 1.

Computation/memory per iteration. In Algorithm 1, the cost of each iteration relates to updating two coordinate blocks. Notably, Steps 2-3 is exactly the standard SGD. Thus the overhead of our method comes from Steps 4-5. We compare our improved EigenPro to the original EigenPro and to standard SGD in Table 1. We see that the overhead of original EigenPro (in bold) scales with the data size n. In contrast, improved EigenPro depends only on the fixed coordinate block size s which is independent of n. Hence, when n becomes large, the overhead of our iteration becomes negligible (both in computation and memory) compared to the cost of SGD.

	Computation	Memory
Improved EigenPro	$\boldsymbol{s \cdot mq} + n \cdot m(d+l)$	$\boldsymbol{s \cdot q} + n \cdot (m+d+l)$
Original EigenPro	$\boldsymbol{n \cdot mq} + n \cdot m(d+l)$	$\boldsymbol{n \cdot q} + n \cdot (m+d+l)$
SGD	$n \cdot m(d+l)$	$n \cdot (m+d+l)$

Table 1: Overhead over SGD is **bolded**. n: training data size, m: batch size, d: feature dim., s: fixed coordinate block size, q: EigenPro parameter, l: number of labels.

To give a realistic example, for many of our experiments $n = 10^6$, while s is chosen to be 10^4. We typically have d, m of the same order of magnitude 10^3, while q and l around 10^2. This results in overhead of EigenPro of less than 1% over SGD for both computation and memory.

5 Experimental Evaluation

Computing resource. We run all experiments on a single workstation equipped with 128GB main memory, two Intel Xeon(R) E5-2620 processors, and one Nvidia GTX Titan Xp (Pascal) GPU.

Datasets. We reduce multiclass labels to multiple binary labels. For image datasets including MNIST (LeCun et al., 1998), CIFAR-10 (Krizhevsky & Hinton, 2009), and SVHN (Netzer et al., 2011), color images are first transformed to grayscale images. We then rescale the range of each feature to $[0, 1]$. For ImageNet (Deng et al., 2009), we use the top 500 PCA components of some convolutional features extracted from Inception-ResNet-v2 (Szegedy et al., 2017). For TIMIT (Garofolo et al., 1993), we normalize each feature by z-score.

Choosing the size of the fixed coordinate block s. We choose s according to the size of the training data, n. When $n \leq 10^5$, we choose $s = 2 \cdot 10^3$; when $n > 10^5$, we choose $s = 1.2 \cdot 10^4$.

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

In Table 2, 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.

Among all the compared methods, FALKON (Rudi et al., 2017) and EigenPro (Ma & Belkin, 2017) stand out for their competitive performance and fast training on a single GPU. Notably, our method still achieves 5X-6X acceleration over FALKON and 5X-14X acceleration over EigenPro with mostly better accuracy. Importantly, our method has the advantage of automatically inferring parameters for optimization. In contrast, parameters related to optimization for FALKON and EigenPro need to be selected by cross-validation.

5.2 Convergence comparison to SGD and EigenPro

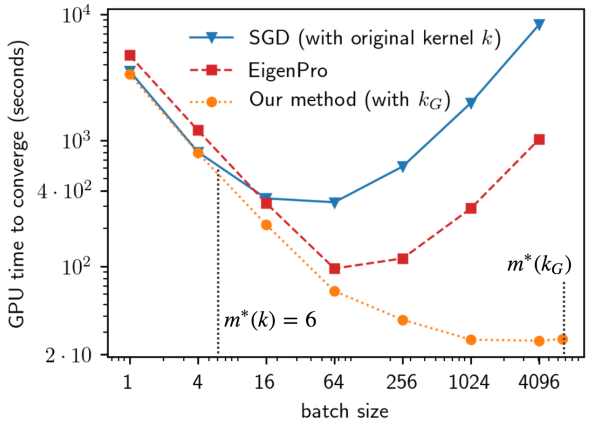
In Figure 2, we train three kernel machines with EigenPro 2.0, standard SGD and EigenPro (Ma & Belkin, 2017) for various batch sizes. The step sizes for SGD and EigenPro are tuned for best performance. The step size for EigenPro 2.0 is computed automatically according to Section 3.

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, our adaptive 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 Table 1).

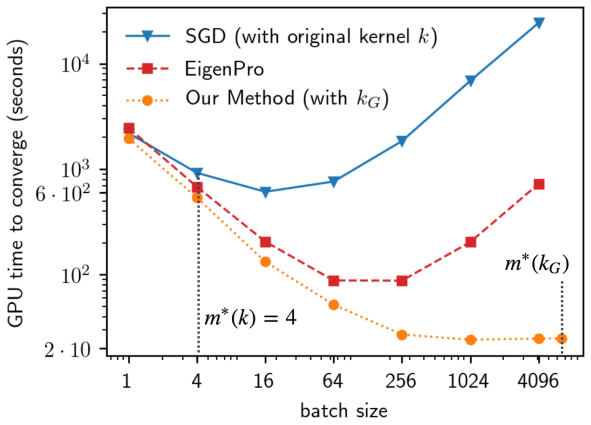
Table 2: Comparison of EigenPro 2.0 and state-of-the-art kernel methods

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 (Ma & Belkin, 2017)
				1.1 h on 1344 AWS vCPUs	0.72%	PCG (Avron et al., 2016)
				less than 37.5 hours on 1 Tesla K20m	0.85%	(Lu et al., 2014)
ImageNet†	1.3×10^6	20.6%	**40 m**	-	19.9%	Inception-ResNet-v2 (Szegedy et al., 2017)
				4 h on 1 Tesla K40c	20.7%	FALKON (Rudi et al., 2017)
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 (Ma & Belkin, 2017)
				1.5 h on 1 Tesla K40c	32.3%	FALKON (Rudi et al., 2017)
				512 IBM Blue Gene/Q cores	33.5%	Ensemble (Huang et al., 2014)
		32.1%	**8 m** (1 epoch)	7.5 h on 1024 AWS vCPUs	33.5%	BCD (Tu et al., 2016)
				multiple AWS g2.2xlarge instances	32.4%	DNN (May et al., 2017)
				multiple AWS g2.2xlarge instances	30.9%	SparseKernel (May et al., 2017) (use learned features)
SUSY	$4 \cdot 10^6$	19.7%	**58 s**	6 m on 1 GTX Titan X	19.8%	EigenPro (Ma & Belkin, 2017)
				4 m on 1 Tesla K40c	19.6%	FALKON (Rudi et al., 2017)
				36 m on IBM POWER8	$\approx 20\%$	Hierarchical (Chen et al., 2016)

† 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 (Szegedy et al., 2017) and show nearly identical performance.
‡ There are two sampling rates for TIMIT, which result in two training sets of different sizes.



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

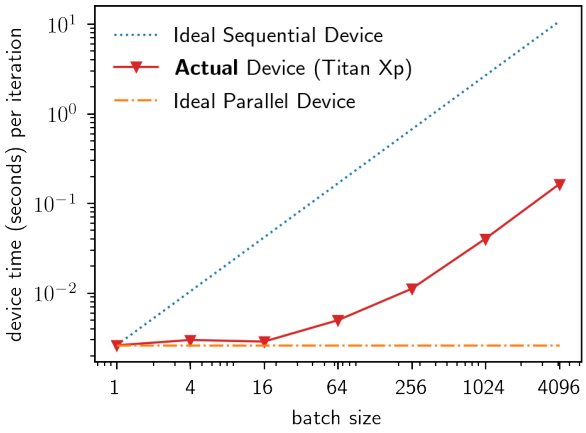


(b) TIMIT (10^5 subsamples), stop when train mse $< 2 \cdot 10^{-4}$

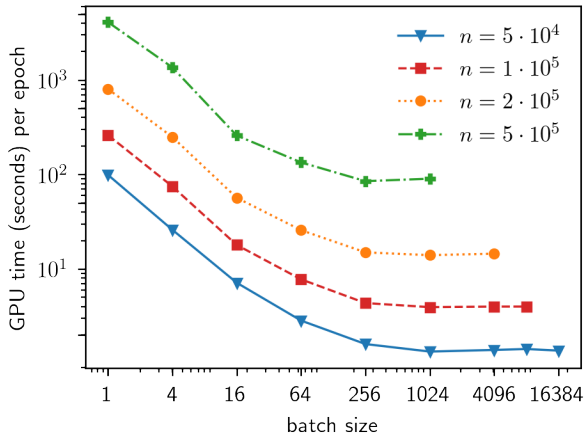
Figure 2: Time to converge with different batch sizes and optimal step sizes

5.3 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



(a) Time per training iteration of 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 for training with different batch sizes

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.

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 (Rodgers, 1985). 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.

5.4 "Interactive" training for exploratory machine learning

Dataset	Size	Feature	EigenPro (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 3: Comparing training time of kernel machines

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 3). 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 (Chang & Lin, 2011) and ThunderSVM (Wen et al., 2018), a fast GPU implementation for LibSVM. We show the results for LibSVM [4] 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.[5] 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.

[4] We use the svm package in scikit-learn 0.19.0.

[5] Our algorithm is still much faster than LibSVM when running on CPU. For example, training on datasets shown in Table 3 takes between one and three minutes.

5.5 Practical Techniques for Accelerating Inference

We would like to point out two simple and practical techniques to accelerate and simplify kernel training. The use of the Laplacian kernel is not common in the literature and in our opinion deserves more attention. While PCA is frequently used to speed up training (and sometimes to improve the test results), it is useful to state the technique explicitly.

Choice of kernel function. In many cases Laplace (exponential) kernel $k(\boldsymbol{x}, \boldsymbol{z}) = e^{-\frac{\|\boldsymbol{x}-\boldsymbol{z}\|}{\sigma}}$ produces results comparable or better than those for the more standard Gaussian kernel. Moreover the Laplacian kernel has several practical advantages over the Gaussian (consistent with the findings reported in (Belkin et al., 2018)). (1) Laplacian generally requires fewer epochs for training to obtain the same quality result. (2) The batch value m^* is typically larger for the Laplacian kernel allowing for more effective parallelization. (3) Test performance for the Laplacian kernel is empirically more robust to the bandwidth parameter σ, significantly reducing the need for careful parameter tuning to achieve optimal performance.

Dimensionality reduction by PCA. Recall that the primary cost of one EigenPro iteration is $n \cdot md$ for the number of operations and $n \cdot (m + d)$ for memory where d is the number of features. Thus reducing the dimension of the features results in significant computational savings. It is often possible to significantly reduce dimensionality of the data without perceptibly changing classification (or regression) accuracy by applying the Principal Components Analysis (PCA). For example, using PCA to reduce the feature dimensionality from 1536 to 500 for ImageNet decreases the accuracy by less than 0.2%.

6 Conclusion and Future Directions

Best practices for training modern large-scale models are concerned with linear scaling. Most of the work is based on an implicit but widely held assumption that the limit of linear scaling, m^*, cannot be controlled or changed in practice. In contrast, this paper shows that the limit of linear scaling can be analytically and automatically adapted to a given computing resource. This finding adds a new dimension for potential improvements in training large-scale models.

The main technical contribution of this paper is a new learning framework (EigenPro 2.0) that extends linear scaling to match the parallel capacity of a computational resource. The framework is based on extracting limited second order information to modify the optimization procedure without changing the learned predictor function. While our paper deals with kernel machines, similar ideas are applicable to a much broader class of learning architectures including deep neural networks.

The algorithms developed in this paper allow for very fast kernel learning on smaller datasets and easy scaling to several million data points using a modern GPU. It is likely that more effective memory management together with better 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 setups is the next natural step for kernel methods.

Acknowledgements

We thank Raef Bassily for discussions and helpful comments and Alex Lee for running ThunderSVM comparisons. We thank Lorenzo Rosasco and Luigi Carratino for sharing preprocessed ImageNet features. We used a Titan Xp GPU provided by Nvidia. We acknowledge financial support from NSF.

References

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

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

Belkin, M., Ma, S., and Mandal, S. To understand deep learning we need to understand kernel learning. *arXiv preprint arXiv:1802.01396*, 2018.

Chang, C.-C. and Lin, C.-J. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27, 2011.

Chen, J., Avron, H., and Sindhwani, V. Hierarchically compositional kernels for scalable nonparametric learning. *arXiv preprint arXiv:1608.00860*, 2016.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 248–255. IEEE, 2009.

Garofolo, J. S., Lamel, L. F., Fisher, W. M., Fiscus, J. G., and Pallett, D. S. Darpa timit acoustic-phonetic continous speech corpus cd-rom. *NIST speech disc*, 1-1.1, 1993.

Golmant, N., Vemuri, N., Yao, Z., Feinberg, V., Gholami, A., Rothauge, K., Mahoney, M. W., and Gonzalez, J. On the computational inefficiency of large batch sizes for stochastic gradient descent. *arXiv preprint arXiv:1811.12941*, 2018.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Huang, P.-S., Avron, H., Sainath, T. N., Sindhwani, V., and Ramabhadran, B. Kernel methods match deep neural networks on timit. In *ICASSP*, pp. 205–209. IEEE, 2014.

Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., Xie, L., Guo, Z., Yang, Y., Yu, L., et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

Krizhevsky, A. and Hinton, G. Learning multiple layers of features from tiny images. Master's thesis, University of Toronto, 2009.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pp. 2278–2324, 1998.

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

Lu, Z., May, A., Liu, K., Garakani, A. B., Guo, D., Bellet, A., Fan, L., Collins, M., Kingsbury, B., Picheny, M., and Sha, F. How to scale up kernel methods to be as good as deep neural nets. *arXiv preprint arXiv:1411.4000*, 2014.

Ma, S. and Belkin, M. Diving into the shallows: a computational perspective on large-scale shallow learning. In *Advances in Neural Information Processing Systems*, pp. 3781–3790, 2017.

Ma, S., Bassily, R., and Belkin, M. The power of interpolation: Understanding the effectiveness of sgd in modern over-parametrized learning. *arXiv preprint arXiv:1712.06559*, 2017.

May, A., Garakani, A. B., Lu, Z., Guo, D., Liu, K., Bellet, A., Fan, L., Collins, M., Hsu, D., Kingsbury, B., et al. Kernel approximation methods for speech recognition. *arXiv preprint arXiv:1701.03577*, 2017.

McCandlish, S., Kaplan, J., Amodei, D., and Team, O. D. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop*, volume 2011, pp. 4, 2011.

Rodgers, D. P. Improvements in multiprocessor system design. In *SIGARCH*, 1985.

Rudi, A., Carratino, L., and Rosasco, L. Falkon: An optimal large scale kernel method. In *Advances in Neural Information Processing Systems*, pp. 3891–3901, 2017.

Smith, S. L., Kindermans, P.-J., and Le, Q. V. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, pp. 12, 2017.

Takác, M., Bijral, A. S., Richtárik, P., and Srebro, N. Mini-batch primal and dual methods for SVMs. In *ICML (3)*, pp. 1022–1030, 2013.

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

Wen, Z., Shi, J., Li, Q., He, B., and Chen, J. Thundersvm: a fast svm library on gpus and cpus. *The Journal of Machine Learning Research (JMLR)*, 19(1):797–801, 2018.

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

You, Y., Gitman, I., and Ginsburg, B. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

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

Appendices

A Datasets

We reduce multiclass labels to multiple binary labels. For image datasets including MNIST (LeCun et al., 1998), CIFAR-10 (Krizhevsky & Hinton, 2009), and SVHN (Netzer et al., 2011), color images are first transformed to grayscale images. We then rescale the range of each feature to $[0, 1]$. For ImageNet (Deng et al., 2009), we use the top 800 PCA components of some convolutional features extracted from Inception-ResNet-v2 (Szegedy et al., 2017). For TIMIT (Garofolo et al., 1993), we normalize each feature by z-score.

B Selection of Kernel and its Bandwidth

We use Gaussian kernel $k(x, y) = \exp(-\frac{\|x-y\|^2}{2\sigma^2})$ and Laplace kernel $k(x, y) = \exp(-\frac{\|x-y\|}{\sigma})$ in our experiments. Note that the kernel bandwidth σ is selected through cross-validation on a small subsampled dataset. In Table 4, we report the kernel and its bandwidth selected for each dataset to achieve the best performance. We also report the parameters that are calculated automatically using our method. Note that in practice we choose a value q (in the parenthesis) that is larger than the q corresponding to $m_{\mathcal{G}}$. Increasing q appears to lead to faster convergence. We use a simple heuristic to automatically obtain such q based on the eigenvalue and the size of the fixed coordinate block[6].

C Analysis of Acceleration

Claim (Acceleration). Using the adaptive kernel $k_{\mathcal{G}}$ decreases the resource time required for training over the original kernel k by a factor of $a \approx \frac{\beta(K)}{\beta(K_{\mathcal{G}})} \cdot \frac{m_{\mathcal{G}}^{max}}{m^*(k)}$.

We will now give a derivation of this acceleration factor a, based on the analysis of SGD in the interpolating setting in (Ma et al., 2017).

As before, let $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$ be the data, and let K be the corresponding (normalized) kernel matrix $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)/n$. We start by recalling the SGD iteration in the kernel setting for a mini-batch of size m, $(\boldsymbol{x}_{t_1}, y_{t_1}), \ldots, (\boldsymbol{x}_{t_m}, y_{t_m})$,

$$f \leftarrow f - \eta \cdot \frac{2}{m} \left\{ \sum_{i=1}^{m} (f(\boldsymbol{x}_{t_i}) - y_{t_i}) k(\boldsymbol{x}_{t_i}, \cdot) \right\}$$

[6]For SUSY we directly specify a large q for optimal performance.

When step size η is chosen optimally, we can apply Theorem 4 in (Ma et al., 2017) to bound its convergence per iteration toward the optimal (interpolating) solution f^* as follows:

$$\mathbb{E}\left[\|f_t - f^*\|_{\mathscr{K}}^2\right] \leq g_K^*(m) \cdot \mathbb{E}\left[\|f_{t-1} - f^*\|_{\mathscr{K}}^2\right]$$

Here $g_K^*(m)$ is a kernel-dependent upper bound on the convergence rate.

The fastest (up to a small constant factor) convergence rate per iteration is obtained when using mini-batch size $m^*(K) = \frac{\beta(K)}{\lambda_1(K) - \lambda_n(K)}$ (or larger). Kernels used in practice, such as Gaussian kernels, have rapid eigendecay (Ma & Belkin, 2017), i.e., $\lambda_1(K) \gg \lambda_n(K)$. Hence we have $m^*(k) \approx \frac{\beta(K)}{\lambda_1(K)}$.

Thus we can write an accurate approximation of convergence rate $g_K^*(m^*(K))$ as follows:

$$\begin{aligned} \epsilon_K^* \triangleq g_K^*(m^*(K)) &= 1 - \frac{m^*(K) \cdot \lambda_n(K)}{\beta(K) + (m-1)\lambda_n(K)} \\ &\approx 1 - \frac{\frac{\lambda_n(K)}{\lambda_1(K)}}{1 + (m-1)\frac{\lambda_n(K)}{\beta(K)}} \end{aligned}$$

We now observe that $\beta = \max_{i=1,\ldots,n} k(\boldsymbol{x}_i, \boldsymbol{x}_i) \geq \mathsf{tr}(K)$. Hence for the mini-batch size m much smaller than n we have

$$(m-1)\frac{\lambda_n(K)}{\beta(K)} \leq (m-1)\frac{\lambda_n(K)}{\mathsf{tr}(K)} \leq \frac{m-1}{n} \ll 1$$

That allows us to write

$$\epsilon_K^* \approx 1 - \frac{\lambda_n(K)}{\lambda_1(K)}$$

We will now apply this formula to the adaptive kernel $k_{\mathcal{G}}$. Recall that its corresponding kernel matrix $K_{\mathcal{G}}$ modifies the top-q eigenspectrum of K such that

$$\lambda_i(K_{\mathcal{G}}) = \begin{cases} \lambda_q(K) & \text{if } i \leq q \\ \lambda_i(K) & \text{if } i > q \end{cases}$$

Thus the convergence rate for $k_{\mathcal{G}}$ is

$$\epsilon_{K_{\mathcal{G}}}^* \approx 1 - \frac{\lambda_n(K_{\mathcal{G}})}{\lambda_1(K_{\mathcal{G}})} = 1 - \frac{\lambda_n(K)}{\lambda_q(K)}$$

Next, we compare the number of iterations needed to converge to error ϵ using the original kernel k and the adaptive kernel $k_{\mathcal{G}}$.

First, we see that for kernel k it takes $t = \frac{\log \epsilon}{\log \epsilon_K^*}$ iterations to go below error ϵ such that

$$\mathbb{E}\left[\|f_t - f^*\|_{\mathscr{K}}^2\right] \leq \epsilon \cdot \mathbb{E}\left[\|f_0 - f^*\|_{\mathscr{K}}^2\right]$$

Table 4: Selected kernel bandwidth and corresponding optimization parameters

Dataset	Size of (Subsampled) Train Set	Kernel	Bandwidth	Train epochs	Calculated Parameters		
					q (adjusted q)	$m = m_{\mathcal{G}}$	η
MNIST	$1 \cdot 10^6$	Gaussian	5	4	93 (330)	735	379
TIMIT	$1.1 \cdot 10^6$	Laplacian	15	3	52 (128)	682	343
ImageNet	$1.3 \cdot 10^6$	Gaussian	16	1	2 (321)	294	149
SUSY	$6 \cdot 10^5$	Gaussian	4	1	106 (850)	1687	849

Notice that $\lambda_n(K) \leq \frac{\mathsf{tr}(K)}{n} = \frac{1}{n}$ for normalized kernel matrix K. Thus for large n, we have

$$\frac{\log \epsilon}{\log \epsilon_K^*} = \frac{\log \epsilon}{\log\left(1 - \frac{\lambda_n(K)}{\lambda_1(K)}\right)} \approx \log \epsilon \cdot \frac{\lambda_1(K)}{\lambda_n(K)}$$

In other words, the number of iterations needed to converge with kernel k is proportional to $\frac{\lambda_1(K)}{\lambda_n(K)}$.

By the same token, to achieve accuracy ϵ, the adaptive kernel $k_{\mathcal{G}}$ needs $\frac{\log \epsilon}{\log \epsilon_{K_{\mathcal{G}}}^*} \approx \log \epsilon \cdot \frac{\lambda_q(K)}{\lambda_n(K)}$ iteration.

Therefore, to achieve accuracy ϵ, training with the adaptive kernel $k_{\mathcal{G}}$ needs $\frac{\lambda_q(K)}{\lambda_1(K)}$ as many iterations as training with the original kernel k.

To unpack the meaning of the ratio $\frac{\lambda_q(K)}{\lambda_1(K)}$, we rewrite it as

$$\frac{\lambda_q(K)}{\lambda_1(K)} = \frac{\lambda_1(K_{\mathcal{G}})}{\lambda_1(K)} = \frac{\beta(K_{\mathcal{G}})}{\beta(K)} \cdot \frac{m^*(K)}{m^*(K_{\mathcal{G}})} = \frac{\beta(K_{\mathcal{G}})}{\beta(K)} \cdot \frac{m^*(K)}{m_{\mathcal{G}}^{max}}$$

Recall that by the assumptions made in the paper (1) any iteration for kernel K with mini-batch size $m \leq m_{\mathcal{G}}^{max}$ requires the same amount of resource time to complete on $\mathcal{G}$, (2) iteration of kernels K and $K_{\mathcal{G}}$ require the same resource time for any m (negligible overhead).

Since $m^*(K) \leq m^*(K_{\mathcal{G}}) \approx m_{\mathcal{G}}^{max}$, we see that one iteration of batch size $m^*(K)$ and one iteration of batch size $m^*(K_{\mathcal{G}})$ take the same amount of time for either kernel.

We thus conclude that the adaptive kernel accelerates over the original kernel by a factor of approximately

$$\frac{\beta(K)}{\beta(K_{\mathcal{G}})} \cdot \frac{m_{\mathcal{G}}^{max}}{m^*(K)}$$

Remark. Notice that our analysis is based on using upper bounds for convergence. While these bounds are tight ((Ma et al., 2017), Theorem 3), there is no guarantees of tightness for specific data and choice of kernel used in practice. Remarkably, the values of parameters obtained by using these bounds work very well in practice. Moreover, acceleration predicted theoretically closely matches acceleration observed in practice.

D. Artifact Appendix

D.1 Abstract

This artifact contains the tensorflow implementation of EigenPro 2.0 (from github.com/EigenPro/EigenPro2) and a python script for running examples using public datasets. It can validate the functionality of our method and support the result in Table 2 of our SysML'2019 paper: **Learning kernels that adapt to GPUs**.

D.2 Artifact check-list (meta-information)

- **Algorithm:** EigenPro iteration.
- **Program:** Python code.
- **Data set:** Public available image datasets.
- **Run-time environment:** Ubuntu 16.04 with CUDA ($\geq$ 8.0) and GPU Computing SDK installed.
- **Hardware:** Any GPU with compute capacity $\geq$ 3.0 (tested GPU: Nvidia Titan Xp (Pascal)).
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT License.
- **Archived (provide DOI)?:** doi.org/10.5281/zenodo.2574996

D.3 Description

D.3.1 How delivered

EigenPro 2.0 is an open source library under MIT license and is hosted with code, API specifications, usage instructions, and design documentations on Github.

D.3.2 Hardware dependencies

EigenPro 2.0 requires NVIDIA GPU with compute capacity $\geq$ 3.0.

D.3.3 Software dependencies

EigenPro 2.0 requires CUDA ($\geq$ 8.0), Tensorflow ($\geq$ 1.2.1), and Keras (tested version: 2.0.8). EigenPro 2.0 has been tested on Ubuntu 16.04 and Windows 10.

D.3.4 Data sets

All datasets are publicly available. The used dataset MNIST in this artifact will be automatically downloaded and preprocessed by the included python script. Users can also download the dataset directly from yann.lecun.com/exdb/mnist.

D.4 Installation

The python based EigenPro 2.0 can be used directly out of the package.

D.5 Experiment workflow

Below are the steps to download and run the experiments.

- Download the code from Github.

```
$ git clone \
> https://github.com/EigenPro/EigenPro2.git
$ cd EigenPro2
```

- Run the test code. Note that the value of mem_gb needs to be the size of the available GPU memory.

```
$ python run_mnist.py --kernel=Gaussian \
> --s=5 --mem_gb=12 --epochs 1 2 3 4 5
```

D.6 Evaluation and expected result

The expected results include automatically calculated hyperparameters for optimization and runtime, as well as classification error (in %) and mean squared error (l2) for both training set and test set (val in the result).