Efficient Second-Order Optimization for Neural Networks with Kernel Machines

Yawen Chen
South China University of Technology
Guangzhou, China
ywchenscut@gmail.com

Yile Chen
South China University of Technology
Guangzhou, China
jireh.x6@gmail.com

Jian Chen
South China University of Technology
Guangzhou, China
ellachen@scut.edu.cn

Zeyi Wen*
Hong Kong University of Science and Technology (Guangzhou); Hong Kong University of Science and Technology Guangzhou & Hong Kong SAR, China
wenzeyi@ust.hk

Jin Huang
South China Normal University
Guangzhou, China
huangjin@m.scnu.edu.cn

ABSTRACT

Second-order optimization has been recently explored in neural network training. However, the recomputation of the Hessian matrix in the second-order optimization posts much extra computation and memory burden in the training. There have been some attempts to address this issue by approximation on the Hessian matrix, which unfortunately degrades the performance of the neural models. In order to tackle this issue, we propose Kernel Stochastic Gradient Descent (Kernel SGD) which solves the optimization problem in a space transformed by the Hessian matrix of the kernel machine. Kernel SGD eliminates the Hessian matrix recomputation in the training and requires a much smaller memory cost which can be controlled via the mini-batch size. We show that Kernel SGD optimization is theoretically guaranteed to converge. Our experimental results on tabular, image and text data confirm that Kernel SGD converges up to 30 times faster than the existing second-order optimization techniques, and achieves the highest test accuracy on all the tasks tested. Kernel SGD even outperforms the first-order optimization baselines in some problems tested in our experiments.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; **Kernel methods**; • **Theory of computation** → *Preconditioning*.

KEYWORDS

neural networks, kernel machines, second-order optimization

ACM Reference Format:
Yawen Chen, Yile Chen, Jian Chen, Zeyi Wen, and Jin Huang. 2022. Efficient Second-Order Optimization for Neural Networks with Kernel Machines. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management (CIKM '22), October 17–21, 2022, Atlanta, GA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3511808.3557307

*Correspondence to Zeyi Wen at wenzeyi@ust.hk.

CIKM '22, October 17–21, 2022, Atlanta, GA, USA

ACM ISBN 978-1-4503-9236-5/22/10…$15.00
https://doi.org/10.1145/3511808.3557307

1 INTRODUCTION

A recent trend in training neural networks is to exploit the second-order information which helps escape the ill-conditioned loss region and boosts the convergence [6, 31, 32]. Such methods use the second-order derivatives on the weights (i.e., edges or connections) of neural networks, and all the second-order derivatives together form a Hessian matrix for each layer. The repeated computation of the Hessian matrix posts much extra computation and memory burden in the training, as the Hessian matrix which is computed based on the neural network weights needs to be updated in each training iteration. The computation and memory burden become huge for large networks, as the size of the Hessian matrix is quadratic in the number of weights. Researchers attempted to improve the efficiency and memory cost of the second-order optimization with approximation such as quasi-Newton [15], Fisher information [19] and diagonalization [6]. Nonetheless, the approximation may lead to the degradation on the model performance such as the predictive accuracy [31].

Kernel machines have achieved comparable performance with neural networks when solving some machine learning problems. For example, a recent study [29] on a popular sentiment analysis problem shows that SVM based solutions can achieve competitive predictive accuracy to the deep neural network based approaches [7]. Belkin et al. [1] demonstrated that kernel machines can fit the problem with random labels easily and produce robust generalization comparable to neural networks. As kernel machines take advantage of convexity and have used second-order optimization in the training [9], incorporation of the second-order information from convex kernel machine problems into deep learning problems may better guide the optimization. Based on this inspiration, we exploit the second-order information from kernel machines and propose an efficient second-order optimization method–Kernel Stochastic Gradient Descent (hereafter "Kernel SGD") for neural network training.

Kernel SGD solves the neural network optimization problem in a space transformed by the Hessian matrix of the kernel machine,

where the training may converge to a better solution. The Hessian matrix in Kernel SGD is proportional to the size of the mini-batch of training instances, rather than the number of weights. As a result, the size of the Hessian matrix in Kernel SGD can be controlled by setting the size of the mini-batch. Another important property of Kernel SGD is that the Hessian matrix does not need to be updated in each iteration, because it is computed based on the training instances which are unchanged during backward and forward propagation. To summarize, our main contributions in this paper are listed as follows.

- We propose Kernel SGD which incorporates the second-order information of kernel machines into the training of neural networks. Kernel SGD exploits the Hessian matrix of the kernel machine which is proportional to the size of mini-batch, and the size of which is controllable by practitioners. The recomputation of Hessian matrix is eliminated benefiting from the unchanged information in the Hessian matrix.
- We theoretically prove that the optimization using Kernel SGD is guaranteed to converge and our theoretical analysis indicates that Kernel SGD is more likely to converge to a better solution.
- We conduct extensive experiments on tabular, image and text data to investigate the behaviours of Kernel SGD. Our experimental results confirm that Kernel SGD converges up to 30 times faster, in comparison with the existing second-order optimization. Kernel SGD can achieve the highest test accuracy on all the tasks evaluated. Especially on *imdb* data set, Kernel SGD improves the test accuracy by around 13%. The memory cost for Kernel SGD is much smaller especially with larger neural networks. As a sanity check, we also compare Kernel SGD with the first-order optimization. Our results show that Kernel SGD even outperforms the first-order optimization in terms of accuracy and convergence time in some problems tested.

2 PRELIMINARIES ON KERNEL MACHINES

A kernel machine projects a non-linear problem into a feature space where the problem may be linearly separable [11, 13]. Formally, suppose we have a training data set $\{X, \boldsymbol{y}\}$. The data set consists of n training instances where $\{X \in \mathbb{R}^{n\times d}, \boldsymbol{y} \in \mathbb{R}^n\} = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), ..., (\boldsymbol{x}_n, y_n)\}$, and $(\boldsymbol{x}_i, y_i)$ denotes the instance $\boldsymbol{x}_i \in \mathbb{R}^d$ with its label y_i. The objective of kernel machine training is to find an optimal weight vector $\boldsymbol{\omega}$ which minimizes the structural risk as follows.

$$\min L(\boldsymbol{\omega}) = \frac{1}{n}\sum_{i=1}^{n} l(f(\boldsymbol{\omega}, \boldsymbol{x}_i), y_i) + \frac{\lambda}{2}||\boldsymbol{\omega}||^2, \tag{1}$$

where λ denotes the regularization constant. The decision function $f(\boldsymbol{\omega}, \boldsymbol{x}_i)$ is computed as $f(\boldsymbol{\omega}, \boldsymbol{x}_i) = \langle \boldsymbol{\omega}, \phi(\boldsymbol{x}_i)\rangle$ where $\boldsymbol{\omega}$ is defined on the *reproducing kernel Hilbert space* (RKHS) and $\langle\cdot,\cdot\rangle$ is the inner product on RKHS. The function $\phi(\cdot)$ maps the instances from their original data space to a higher dimensional feature space induced by the kernel function. Assume the loss $l(\cdot,\cdot)$ is an affine function of $\boldsymbol{\omega}$ as the affine loss function is widely used in kernel machines (e.g., SVMs) and is robust to the outliers [3]. Since Problem (1) is an instance of the *representer theorem* [25], we can derive that a minimizer of Problem (1) is $\boldsymbol{\omega} = \sum_{j=1}^{n} \alpha_j \phi(\boldsymbol{x}_j)$. Based on the *reproducing property* [27], we have $f(\boldsymbol{\omega}, \boldsymbol{x}_i) = \sum_{j=1}^{n} \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ where $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is a positive-definite kernel and $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \langle \phi(\boldsymbol{x}_i), \phi(\boldsymbol{x}_j)\rangle$. By substituting the expressions of $f(\boldsymbol{\omega}, \boldsymbol{x}_i)$ and $\boldsymbol{\omega}$ into Problem (1), we rewrite the objective with respect to $\boldsymbol{\alpha}$ below.

$$\min L(\boldsymbol{\alpha}) = \frac{1}{n}\sum_{i=1}^{n} l(\sum_{j=1}^{n} \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j), y_i) + \frac{\lambda}{2}||\sum_{j=1}^{n} \alpha_j \phi(\boldsymbol{x}_j)||^2, \tag{2}$$

where $\boldsymbol{\alpha} = [\alpha_1 \ \dots \ \alpha_n]^T$ is an n-dimension vector, each dimension of which corresponds to the contribution of a training instance to the kernel machine.

Next we compute the Hessian matrix of Problem (2). We discuss two situations where Problem (2) is solved with or without constraints. As it is easy to compute the Hessian matrix of Problem (2) without any constraints, we concentrate on the constrained problem in the following. The Hessian matrix of unconstrained Problem (2) is equal to the one of constrained Problem (2). Problem (2) with constraints can be written as follows.

$$\begin{aligned} &\min L(\boldsymbol{\alpha}) = \frac{1}{n}\sum_{i=1}^{n} l(\sum_{j=1}^{n} \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j), y_i) + \frac{\lambda}{2}||\sum_{j=1}^{n} \alpha_j \phi(\boldsymbol{x}_j)||^2, \\ &\text{subject to } \lambda > 0, \\ &\qquad h_i(X, \boldsymbol{\alpha}, \Theta) = 0, \forall i \in \{1, \dots, n_h\}, \\ &\qquad g_j(X, \boldsymbol{\alpha}, \Theta) \le 0, \forall j \in \{1, \dots, n_g\}, \end{aligned} \tag{3}$$

where Θ is the set of hyper-parameters in kernel machines (i.e., $\Theta = \{\lambda, \theta_1, \theta_2, \dots\}$). The number of equality constraints in the set $\mathcal{H} = \{h_i(\cdot,\cdot,\cdot) | \forall i \in \{1, \dots, n_h\}\}$ and the number of inequality constraints in the set $\mathcal{G} = \{g_j(\cdot,\cdot,\cdot) | \forall j \in \{1, \dots, n_g\}\}$ are denoted by n_h and n_g, respectively. The constraints in $\mathcal{H}$ and $\mathcal{G}$ are affine functions, and constraints in $\mathcal{G}$ are convex and continuously differentiable, which are common in kernel machines such as SVMs. In order to solve the optimization problem as presented in Equation (3), we transform the Problem (3) to the following form with Lagrangian multipliers.

$$\begin{aligned} L(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\mu}) =& \frac{1}{n}\sum_{i=1}^{n} l(\sum_{j=1}^{n} \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j), y_i) + \frac{\lambda}{2}||\sum_{j=1}^{n} \alpha_j \phi(\boldsymbol{x}_j)||^2 \\ &+ \sum_{i=1}^{n_h} \beta_i h_i(X, \boldsymbol{\alpha}, \Theta) + \sum_{j=1}^{n_g} \mu_j g_j(X, \boldsymbol{\alpha}, \Theta). \end{aligned} \tag{4}$$

The transformation is inspired by the proof of Lemma 4 in the paper [13]. Lagrangian multipliers β_i and μ_i denote the i-th element of $\boldsymbol{\beta}$ and $\boldsymbol{\mu}$ respectively where $\boldsymbol{\beta} \in \mathbb{R}^{n_h}$ and $\boldsymbol{\mu} \in \mathbb{R}^{n_g}$. Then, we have the Karush-Kuhn-Tucker (KKT) conditions [13] for the Problem (3) below.

$$\begin{aligned} \frac{\partial L(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\mu})}{\partial \alpha_p} =& \frac{1}{n}\sum_{i=1}^{n} \nabla_{\alpha_p} l(\sum_{j=1}^{n} \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j), y_i) + \lambda \boldsymbol{\alpha}^T K_p \\ &+ \boldsymbol{\beta}^T \frac{\partial \boldsymbol{h}(X, \boldsymbol{\alpha}, \Theta)}{\partial \alpha_p} + \boldsymbol{\mu}^T \frac{\partial \boldsymbol{g}(X, \boldsymbol{\alpha}, \Theta)}{\partial \alpha_p} = 0, \\ \text{subject to} \quad & h_i(X, \boldsymbol{\alpha}, \Theta) = 0, \ \mu_j g_j(X, \boldsymbol{\alpha}, \Theta) = 0, \\ & g_j(X, \boldsymbol{\alpha}, \Theta) \le 0, \ \mu_j \ge 0, \\ & \forall i \in \{1, \dots, n_h\}, \ \forall j \in \{1, \dots, n_g\}. \end{aligned} \tag{5}$$

The n_h dimensional vector function $\boldsymbol{h}(X, \boldsymbol{\alpha}, \Theta)$ treats the constraint $h_i(X, \boldsymbol{\alpha}, \Theta)$ as its i-th dimension; similarly, the vector function $\boldsymbol{g}(X, \boldsymbol{\alpha}, \Theta)$ is formed by $g_j(X, \boldsymbol{\alpha}, \Theta)$ of all j in the set $\{1, \dots, n_g\}$. Let K_p denote the p-th column in the kernel matrix $K \in \mathbb{R}^{n\times n}$. The elements in the i-th row and j-th column of matrix K is defined as $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. Based on the assumptions made earlier, all the terms in the first derivative of the objective $L(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\mu})$ (i.e., the first

two lines in Equation (5)) except for $\lambda \boldsymbol{\alpha}^T K_p$ can be written as a constant with respect to α_p. If we take the second-order derivative of $L(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\mu})$ with respect to α, we can obtain that $\frac{\partial L(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\mu})}{\partial \alpha_p \partial \alpha_q} = K_{pq}$ for all p and q in the set $\{1, \ldots, n\}$. Hence we have the Hessian matrix $H = [H_{ij}]_{n\times n}$ of objective function (5) equal to the kernel matrix K as follows.

$$H = \begin{bmatrix} \frac{\partial L(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\mu})}{\partial \alpha_1 \partial \alpha_1} & \frac{\partial L(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\mu})}{\partial \alpha_1 \partial \alpha_2} & \cdots & \frac{\partial L(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\mu})}{\partial \alpha_1 \partial \alpha_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial L(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\mu})}{\partial \alpha_n \partial \alpha_1} & \frac{\partial L(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\mu})}{\partial \alpha_n \partial \alpha_2} & \cdots & \frac{\partial L(\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\mu})}{\partial \alpha_n \partial \alpha_n} \end{bmatrix}$$
$$= \begin{bmatrix} K_{11} & K_{12} & \cdots & K_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ K_{n1} & K_{n2} & \cdots & K_{nn} \end{bmatrix} = K.$$

The element in the i-th row and j-th column of the matrix H is $H_{ij} = K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. For clarity, we use kernel matrix H to denote the Hessian matrix of the kernel machine in the rest of the paper.

3 OUR PROPOSED KERNEL SGD OPTIMIZATION

In this section, we elaborate our techniques to improve the efficiency of the second-order optimization in neural network training. Exploiting second-order information is non-trivial and needs to address two key challenges. First, the memory and computation cost of second-order optimization is considerably high, due to the large size of the weight matrix and the frequent update of the Hessian matrix in neural network training. Second, simple methods like approximation on the Hessian matrix, although show improvements on memory and computation, may lead to the deficiency in the model performance such as the predictive accuracy.

In order to tackle these challenges, we take advantages of kernel machines and propose an optimization method named "Kernel Stochastic Gradient Descent" (hereafter "Kernel SGD"). Our key inspiration is that kernel machines have achieved remarkable performance as neural networks in some applications [1, 29], while performing convex optimization. Our method integrates the second-order information of kernel machines into the neural network optimization. Benefiting from the unchanged Hessian matrix of the kernel machine, our method shows advantages in computation and space efficiency. Next, we provide technical details in Kernel SGD with theoretical analysis.

3.1 Problem Projection and Update Rule

The key idea of Kernel SGD is to project the optimization problem into another space with the Hessian matrix of kernel machines, where a better solution may be found. Formally, let $\mathcal{J}(W)$ be the objective (e.g., cross entropy loss) of the original neural network problem with respect to the weight matrix $W \in \mathbb{R}^{d_r \times d_c}$. We denote the number of rows and columns in the weight matrix as d_r and d_c, separately. By introducing the projection matrix $P \in \mathbb{R}^{d_c \times d_c}$, we first project the weight matrix W to a new matrix called $\hat{W}$ where $\hat{W} = WP^{\frac{1}{2}}$. The projection matrix, which is a symmetric matrix, is used to project the optimization problem into another space. Thus the problem after projection becomes optimizing the projected loss $\hat{\mathcal{J}}(\hat{W})$ where $\mathcal{J}(W) = \mathcal{J}(\hat{W}P^{-\frac{1}{2}}) = \hat{\mathcal{J}}(\hat{W})$. The standard SGD updates the solution $\hat{W}$ for the projected loss with the equation below.

$$\hat{W}' = \hat{W} - \eta \nabla_{\hat{W}} \hat{\mathcal{J}}(\hat{W}), \tag{6}$$

where η is the learning rate and $\nabla_{\hat{W}} \hat{\mathcal{J}}(\cdot)$ is the gradient of the projected loss $\hat{\mathcal{J}}(\cdot)$ with respect to the weight $\hat{W}$. The dimensions of $\nabla_{\hat{W}} \hat{\mathcal{J}}(\cdot)$ are the same as the dimensions of $\hat{W}$.

In the original neural network problem, the weight matrix to update is W. We expand Equation (6) with the expression of $\nabla_{\hat{W}} \hat{\mathcal{J}}(\hat{W})$ where $\nabla_{\hat{W}} \hat{\mathcal{J}}(\hat{W}) = \nabla_W \mathcal{J}(W)(P^{-\frac{1}{2}})^T$ and derive the following update rule for the weight W.

$$W' = W - \eta \nabla_W \mathcal{J}(W) P^{-1}, \tag{7}$$

where $\nabla_W \mathcal{J}(W) \in \mathbb{R}^{d_r \times d_c}$ is the gradient of loss $\mathcal{J}(\cdot)$ with respect to the weight W. Updating the weight $\hat{W}$ of the projected problem in the transformed space using Equation (6) is equivalent to updating the weight W of the original problem using Equation (7) .

A good choice of projection matrix helps transform the problem into a space where the optimization better converges. In Kernel SGD, we choose the Hessian matrix H of the kernel machine as the projection matrix P. The intuition is that kernel machines and neural networks aim to learn the similar inference function through similar mapping and combination processes, except that kernel machines solve convex problems. Specifically, the kernel machine and neural network first learn a new representation for the instance using mapping (e.g., the mapping with function $\phi(\cdot)$ in kernel machines or the mapping performed by the hidden layers in neural networks) and then the inference function is generated by combining each dimension in the new representation with its weight (e.g., the weight $\boldsymbol{\omega}$ in kernel machines or the weights of the inference layer in neural networks). The kernel matrix may guide the optimization to a better direction in the transformed space, as the kernel matrix contains the second-order information of the convex objective. Hence Kernel SGD updates the weight W with the following rule which corresponds to the update of weight $\hat{W}$ in the space transformed by the kernel matrix.

$$W' = W - \eta \nabla_W \mathcal{J}(W) H^{-1}. \tag{8}$$

When the inverse of the kernel matrix H does not exist, we use the pseudoinverse. In the mini-batch setting, we may not have all the training instances to construct the whole kernel matrix. Thus we use a subset of the training data, for example, a mini-batch of m training instances, to approximate the kernel matrix where H is then an $m \times m$ matrix.

Moreover, the matrix multiplication between the gradient of loss and the inverse kernel matrix constrains that the number of rows in kernel matrix H should equal the number of columns in the gradient $\nabla_W \mathcal{J}(W)$. When the above multiplication constraint is not satisfied, the inverse of kernel matrix is reshaped in the following ways. First, if m is smaller than d_c, we pad the inverse of kernel matrix with elements of zeros so that the number of rows of the kernel matrix satisfies the constraint; Second, if m is greater than d_c, we remove the last $m - d_c$ rows and columns of the inverse of kernel matrix. In Section 3.3, we show that the reshaping of the inverse kernel matrix still guarantees the training converged.

Algorithm 1: Kernel SGD

Input: Training instances X, kernel function k, learning rate η, objective $\mathcal{J}$, and weight $W^{(t)}$ in layer t

1 **foreach** *training iteration* **do**
2 $H \in \mathbb{R}^{m \times m} \leftarrow$ kernel matrix for the current mini-batch
3 // $H_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ where $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are from the current mini-batch
4 $H^{-1} \in \mathbb{R}^{d_c \times d_c} \leftarrow$ the reshaped inverse of H
5 **for** $t \leftarrow \mathcal{T}$ **to** 1 **do** // $\mathcal{T}$ is the total number of layers
6 **if** $t == \mathcal{T}$ **then**
7 Update $W^{(t)}$ using Kernel SGD with Eq. (8)
8 **else**
9 Update $W^{(t)}$ using SGD

3.2 Kernel SGD in Neural Network Training

We show the pseudocode of Kernel SGD in Algorithm 1. Kernel SGD first computes the kernel matrix H for each mini-batch (Line 2). The kernel matrix is stored as an $m \times m$ matrix for each mini-batch and the whole kernel matrix for all the mini-batches is thus an $n \times m$ matrix where n and m are the number of instances and the mini-batch size, respectively. Then we compute the inverse of the kernel matrix for each mini-batch with reshaping if necessary (Line 4). Next, in the backward propagation, Kernel SGD updates the weights of the network from the end to the beginning using Equation (8) with the corresponding inverse matrix. Moreover, the output of the last hidden layer, which can be treated as the learned representation of the input instance, corresponds to the representation $\phi(\boldsymbol{x})$ in kernel machines. The output layer of the network can be regarded as the inference layer which has a similar decision function as general kernel machines. Thus, it is more natural to apply Equation (8) only in the last layer, and the rest of the layers are updated using the original SGD (Line 5-9). We also empirically observed the difference between Kernel SGD applied in the last layer and it applied in all the intermediate layers. The experimental results which are shown in Table 1 further confirm that Kernel SGD applied in the last layer is more computationally efficient and can perform more accurate prediction. Hence the implementation of Kernel SGD follows this manner. More experimental details can be found in Section 4. Note that on *cov-tw* and *imdb*, our method applied in all the layers converges faster due to that the training converges earlier with fewer epochs at the cost of a higher loss.

Unlike the current second-order optimization [2, 32] and preconditioned SGD [6] which all need to update the Hessian matrix in each iteration, Kernel SGD uses fixed kernel matrix. Therefore, the computation cost for kernel matrix is rather small compared with the training of the whole network. We can further accelerate Kernel SGD by computing the kernel matrix in parallel with the training of neural networks. In comparison, updating the Hessian matrix while training the networks is practically infeasible in most second-order optimization, as the Hessian matrix depends on the updated weights. Considering that the mini-batch size is much smaller than the total number of training instances, the storage cost for the kernel matrices of all the mini-batches is also smaller which is linear in the number of instances, i.e., $O(n \cdot m)$.

Table 1: Comparison of Kernel SGD applied to the last layer and Kernel SGD applied to all the layers. The second line indicates the layers that Kernel SGD is applied to.

data set	test accuracy (%)		convergence time (sec.)	
	the last layer	all the layers	the last layer	all the layers
mnist	**97.94±0.30**	95.00 ± 0.39	**272±230**	1030 ± 839
usps	**93.63±0.34**	85.85 ± 0.30	**30±16**	34 ± 1
cifar10	**83.30±0.22**	83.16 ± 0.27	**475±32**	864 ± 239
s-cov	**73.40±1.11**	71.16 ± 1.98	**64±3**	66 ± 2
cov-tw	**77.55±2.07**	74.37 ± 3.61	325 ± 156	**176±82**
imdb	**89.76±0.56**	77.64 ± 9.48	5548 ± 1975	**3908±1233**

3.3 Convergence Analysis

We theoretically demonstrate the convergence guarantee of Kernel SGD optimization. In a general neural network for classification, the last layer (i.e., output layer) is commonly a fully connected (FC) layer followed by a softmax function. We denote the weight matrix of the last layer by $W \in \mathbb{R}^{n_c \times d_h}$ where d_r equals n_c which is the number of classes and d_c equals d_h which is the dimension of the last hidden layer. The output of the last hidden layer is indicated as $G(\boldsymbol{x}) \in \mathbb{R}^{d_h}$ and can be treated as the learned representation of $\boldsymbol{x}$. Let f_i be the i-th input to the last layer. The input f_i is computed with the formula $f_i = W_i G(\boldsymbol{x})$ where W_i is the i-th row in matrix W. Suppose the instance $\boldsymbol{x}$ belongs to the i-th class. The cross entropy loss with respect to W can be defined as follows.

$$\mathcal{J}(W) = -f_i + \ln \sum_{j=1}^{n_c} e^{f_j}. \tag{9}$$

Using the notations above, we give the convergence guarantee of Kernel SGD optimization with the following Theorem.

Theorem 3.1. *Suppose the last layer in the neural network is a fully connected layer with a softmax activation function. Given the weight matrix W of the last layer and the corresponding updated weight matrix W' computed by Equation* (8)*, the cross entropy loss $\mathcal{J}(W)$ and loss $\mathcal{J}(W')$ of the updated weight satisfy the following inequality.*

$$\mathcal{J}(W') \leq \mathcal{J}(W). \tag{10}$$

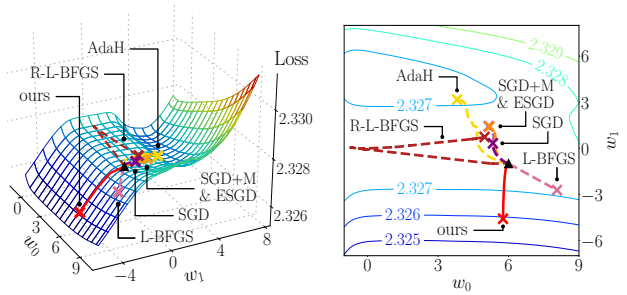
According to Theorem 3.1, the loss decreases or stays unchanged as the training progresses using Kernel SGD. Hence we can conclude that the optimization using Kernel SGD is guaranteed to converge or be terminated if the loss stays unchanged. Next, we propose that Kernel SGD tends to find a better solution in the transformed space.

Proposition 3.2. *Let W_0 be the initial point in the original space and W^* be the minimum of the original loss; Let $\hat{W}^*$ be the minimum of the loss projected by the projection matrix H and $\hat{W}_0$ be the initial point in the projected space. Assume that the eigenvalues of kernel matrix $H \in \mathbb{R}^{m \times m}$ are $\{\pi_1, \ldots, \pi_m\}$ and $\sup(\sum_{i=1}^{m} \pi_i) = \frac{1}{n_c}$. Then we have the following inequality holds.*

$$\left\|\hat{W}^* - \hat{W}_0\right\|_F \leq \left\|W^* - W_0\right\|_F, \tag{11}$$

where $\|\cdot\|_F$ is the Frobenius norm for matrices.

The initial point $\hat{W}_0$ in the transformed space can be treated to be equivalent to W_0 in the original space. Proposition 3.2 indicates that



(a) Landscape of loss (b) Contour lines of loss

Figure 1: Trajectories of loss decrease using different optimization methods. The triangle marker indicates the starting point and the cross markers indicate the ends of the optimization.

the optimum is closer to the initial point in the transformed space. With the same learning rate, Kernel SGD is more likely to find a better solution, thanks to the smaller gap between the optimum and the starting point in the transformed space.

We can get some insight of Proposition 3.2 from observing the trajectories of loss. We trained a shallow network on *usps* data set using Kernel SGD and other optimization methods which are AdaH [32], ESGD [6], L-BFGS [15], R-L-BFGS [2], SGD and SGD with momentum (SGD+M) as introduced in Section 4.1. The network has three linear hidden layers and an FC output layer with the cross entropy loss. For ease of visualization, two randomly selected weights w_0, w_1 were updated in optimization while others were fixed. The loss decrease of trajectories using different optimization methods is depicted in Figure 1. Starting from the same initial point, the optimization using Kernel SGD descends at a better direction and is more likely to converge to a better minimum than the existing methods. More experimental results can be found in Section 4 which further confirm our findings. The proof of Theorem 3.1 and Proposition 3.2 is available in the Appendix.

4 EXPERIMENTAL STUDIES

In this section, we first study the overall performance of Kernel SGD. Then we investigate the influence of different kernel functions and batch sizes on Kernel SGD.

4.1 Data Sets and Experimental Setup

For a better understanding of Kernel SGD on different problems, we conducted the experiments with three major data types: tabular, image and text data. Table 2 shows the details of the data sets. The tabular data sets which include *mnist* and *usps* were downloaded from LIBSVM website[1]. The dimensions of *mnist* and *usps* are 780 and 256, respectively. For the image data, the *cifar10* and *SARS-CoV-2* (*s-cov*) [28] data sets are used. The data set *s-cov* contains RGB CT scans and is used to identify whether the patient is infected by the virus. Each image in the two data sets was reshaped into 32×32 size. The text data sets used are *IMDb* and *COVID-19-tweets* (*cov-tw*) [21]. The data set *cov-tw* collects the English Tweets about COVID-19 and is labeled as informative or not. We took 20% of the training data to serve as the validation set except for *cov-tw*. In *cov-tw* set, 1000 validation instances are provided.

[1] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/

Table 2: Data set information.

type	data set	#training instances	#test instances	#classes
tabular	mnist	60,000	10,000	10
	usps	7,291	2,007	10
image	cifar10	50,000	10,000	10
	s-cov	2,000	482	2
text	cov-tw	7,000	2,000	2
	imdb	25,000	25,000	2

On the tabular data, our method was evaluated using a two-linear-layer neural network which has 100 neurons in each hidden layer. We use the ResNet-18 [10] network for the image classification. To solve text classification problems, we adopt an LSTM network with pre-trained word vectors [23]. The dimensions of the last hidden layers in ResNet-18 and LSTM are 512 and 100 respectively which are the default settings in Pytorch.

Experimental setup: The experiments were conducted on a machine with an Intel(R) Xeon(R) Silver 4210 CPU of 126GB main memory and two GeForce RTX 3090 GPUs running on a Linux OS. We compare our Kernel SGD with the following mainstream second-order optimization methods: (i) ADAHESSIAN (AdaH) [32], (ii) Equilibrated SGD (ESGD) [6], (iii) robust multi-batch L-BFGS (R-L-BFGS) [2] and (iv) L-BFGS [15]. All the baselines and our method are implemented using PyTorch [22]. In Kernel SGD, to compute the kernel matrix, we use the radial basis function (RBF) kernel (i.e., $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\gamma||\boldsymbol{x}_i - \boldsymbol{x}_j||_2^2)$). The learning rate and the hyper-parameter γ for the kernel are selected from $\{1 \times 10^{-1}, 1 \times 10^{-2}, 1 \times 10^{-3}, 1 \times 10^{-4}\}$. The size of history used in L-BFGS and R-L-BFGS is set as 10 for the limited memory. The mini-batch size used is 64. When the change of loss is less than 1×10^{-4} in 3 consecutive epochs or the training reaches 500 epochs, we terminate the training as the training has converged. The best model is selected according to the best performance on the validation set. We repeated each experiment several times to acquire average performance.

4.2 Comparison of Kernel SGD and Second-Order Optimization Baselines

We compare our Kernel SGD with the second-order optimization baseline methods in different aspects to demonstrate the superiority of our method.

Observation on optimization behavior. To study the optimization behavior of Kernel SGD and the second-order optimization baselines, we depict the training loss and validation accuracy varying with epochs in Figure 2 and Figure 3, respectively. Compared with ESGD, L-BFGS, and R-L-BFGS, our Kernel SGD produces the lowest loss and highest validation accuracy. L-BFGS and R-L-BFGS get stuck at the saddle points or local minimums in the early stage. ESGD helps escape the saddle points but fails to converge to a better loss on the tabular and text data. Kernel SGD reaches the

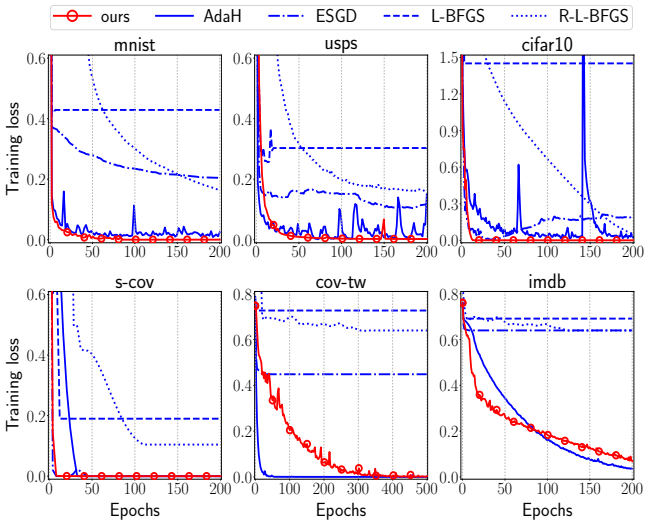


Figure 2: Variations of loss during the training with Kernel SGD and the second-order optimization baselines.

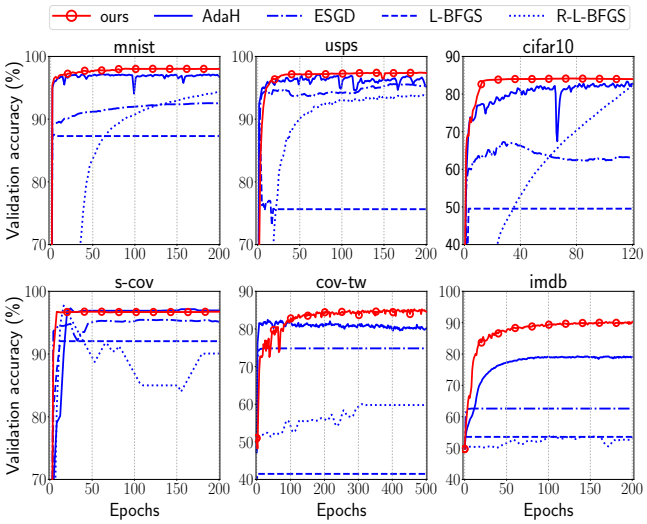


Figure 3: Variations of accuracy during the training with Kernel SGD and the second-order optimization baselines.

stable point in a small number of epochs while achieving prominent results in almost all the tasks. As for AdaH, the loss decreases with significant vibrations. Our method converges more steadily towards a smaller loss compared with AdaH except for the text data. Kernel SGD does not show an advantage of loss over AdaH on the two text data sets (i.e., *cov-tw* and *imdb*). However Kernel SGD yields higher validation accuracy than AdaH. This shows the ability of our method to avoid overfitting by finding a better minimum. Though it seems that AdaH needs fewer epochs than Kernel SGD to converge on the text data, the convergence time of AdaH is, on the contrary, much longer than Kernel SGD as listed in Table 3, for the large computation cost in each epoch.

Analysis on generalization. We investigate the generalization performance by considering the model accuracy. The training (train.) and test accuracy which are achieved by the best model of each optimizer are reported in the third and fourth columns of Table 3. Our method shows a remarkable generalization performance and can achieve the highest test accuracy on all the tasks. Especially on *imdb*, Kernel SGD improves the test accuracy by around 13%. Moreover, the smaller gaps between the test accuracy and training accuracy, as well as the smaller variances, confirm that Kernel SGD can mitigate overfitting and achieves a stable accuracy. For fair comparison, we did not adopt pre-processing techniques on any data sets such as random flipping on the tested images, and hence the accuracy in Table 3 is slightly different from those shown in other studies. We further evaluated *cifar10* with pre-processing using plain SGD and obtained 92.73% accuracy which is similar to the averaging result.

Analysis on convergence speed. We recorded the convergence (converg.) time in the fifth column of Table 3. Kernel SGD converges up to 30 times faster than second-order optimization baselines. On the tabular data, R-L-BFGS converges faster because it stops too early with relatively large losses as shown in Figure 2 and Table 3. Kernel SGD takes more epochs to converge to a better loss and thus needs more convergence time than R-L-BFGS.

Analysis on memory cost. The memory for the Hessian matrix is presented in the last column of Table 3. We show the size of the whole kernel matrix in Kernel SGD which is an n-by-m matrix. In other optimizers, the Hessian matrix of the neural network is not explicitly computed. For L-BFGS and R-L-BFGS, we recorded the storage for the historical information which is used to approximate the Hessian. We computed the memory consumption for the Hessian momentum of AdaH. In ESGD, we recorded the memory of the preconditioning matrix which is equivalent to Hessian. Kernel SGD uses much smaller memory to store the kernel matrix in larger neural networks such as ResNet-18 and LSTM, as the computation is based on training instances rather than the weights in networks.

4.3 Sanity check

Although our main aim in this paper is to improve the second-order optimization methods, for a sanity check and for completeness, we further compare Kernel SGD with first-order optimizers which are mini-batch SGD and SGD with momentum (SGD+M). The momentum hyper-parameter in SGD+M was 0.9 by default. The results are shown in Table 4. Our method still outperforms the first-order optimizers in terms of generality on most tasks (i.e., 5 out of 6 tasks in total). Our method converges even faster than the first-order optimizers in image classification problems.

4.4 Impact of Kernel Function on Kernel SGD

Except for RBF kernel, we evaluated Kernel SGD with the linear kernel and polynomial kernel. The degree hyper-parameter in polynomial kernel is selected from $\{1, 2, 3\}$. We also verified the effectiveness of the kernel matrix by using a random positive semi-definite (PSD) matrix as the projection matrix. From Figure 4, Kernel SGD shows a robust performance over different kernel functions. For

Table 3: Comparison of Kernel SGD and second-order optimization methods in model accuracy (%), convergence time (second) and Hessian matrix size (MB).

data set	optimizer	train. accuracy	test accuracy	converg. time	Hessian
mnist	ours	99.92 ± 0.17	**97.94±0.30**	***272±230***	11.72
	AdaH	99.48 ± 0.53	97.32 ± 0.14	285 ± 78	0.76
	ESGD	93.90 ± 1.04	92.77 ± 0.52	8671 ± 9148	**0.18**
	L-BFGS	87.70 ± 2.86	87.77 ± 2.82	555 ± 535	1.97
	R-L-BFGS	98.29 ± 0.77	96.88 ± 0.42	**103± 2**	4.93
usps	ours	99.90 ± 0.09	**93.63±0.34**	***30±16***	1.42
	AdaH	99.40 ± 0.64	93.46 ± 0.64	32 ± 27	0.36
	ESGD	96.04 ± 0.91	91.72 ± 0.70	37 ± 27	**0.18**
	L-BFGS	93.74 ± 0.81	89.15 ± 1.12	888 ± 838	1.97
	R-L-BFGS	94.35 ± 0.54	89.67 ± 0.76	**14±1**	2.33
cifar10	ours	100.00 ± 0.00	**83.30±0.22**	**475±32**	**9.77**
	AdaH	98.77 ± 0.46	82.11 ± 0.70	11942 ± 4694	48.00
	ESGD	99.79 ± 0.20	67.73 ± 0.38	9758 ± 587	42.63
	L-BFGS	59.53 ± 4.75	55.39 ± 4.13	3894 ± 467	468.92
	R-L-BFGS	98.97 ± 0.81	74.98 ± 0.31	2605 ± 34	554.13
s-cov	ours	100.00 ± 0.00	**73.40±1.11**	**64±3**	**0.39**
	AdaH	99.84 ± 0.33	71.66 ± 3.80	180 ± 60	48.00
	ESGD	99.99 ± 1.42	72.03 ± 2.72	199 ± 270	42.63
	L-BFGS	98.46 ± 0.75	70.00 ± 2.68	607 ± 376	468.92
	R-L-BFGS	62.54 ± 0.01	52.28 ± 0.01	69 ± 1	554.13
cov-tw	ours	90.21 ± 5.21	**77.55±2.07**	**325±156**	**1.71**
	AdaH	98.94 ± 1.81	77.33 ± 1.15	4722 ± 724	22.79
	ESGD	79.68 ± 1.79	71.72 ± 0.78	2590 ± 51	11.40
	L-BFGS	51.84 ± 1.27	52.68 ± 0.43	underfit	125.36
	R-L-BFGS	52.90 ± 0.14	52.64 ± 0.32	underfit	148.14
imdb	ours	95.05 ± 2.75	**89.76±0.56**	**5548±1975**	**4.88**
	AdaH	96.74 ± 1.83	76.28 ± 0.71	63669 ± 38685	22.79
	ESGD	63.68 ± 1.60	61.69 ± 1.89	7116 ± 656	11.40
	L-BFGS	50.99 ± 1.30	50.89 ± 1.25	underfit	125.36
	R-L-BFGS	50.44 ± 0.71	50.48 ± 1.30	underfit	550.10

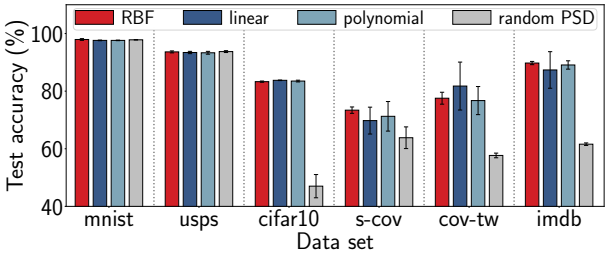


Figure 4: Test accuracy of Kernel SGD with different kernels.

example, Kernel SGD trained on *cifar10* achieved 83.76% and 83.51% test accuracy with linear and polynomial kernels separately, which are similar to that achieved with RBF kernel. The optimization with a random PSD matrix underfits the image data and text data which has only about 50% to 60% accuracy.

4.5 Impact of Batch Size on Kernel SGD

As the kernel matrix is correlated to the data in each mini-batch, we varied the batch size and explore the impact of batch size on

Table 4: Comparison of Kernel SGD and first-order optimization in test accuracy (%) and convergence time (second).

data set	optimizer	test accuracy (%)	converg. time (sec.)
mnist	ours	**97.94±0.30**	272 ± 230
	SGD	97.86 ± 0.07	**130±61**
	SGD+M	97.85 ± 0.21	2832 ± 2849
usps	ours	***93.63±0.34***	30 ± 16
	SGD	93.28 ± 0.20	**27±15**
	SGD+M	**94.17±0.29**	35 ± 28
cifar10	ours	**83.30±0.22**	**475±32**
	SGD	82.59 ± 0.78	644 ± 73
	SGD+M	83.23 ± 0.33	582 ± 19
s-cov	ours	**73.40±0.92**	**64±75**
	SGD	68.13 ± 1.86	71 ± 13
	SGD+M	67.47 ± 3.24	74 ± 10
cov-tw	ours	**77.55±2.07**	325 ± 156
	SGD	74.17 ± 0.18	190 ± 148
	SGD+M	77.03 ± 3.58	**151±98**
imdb	ours	**89.76±0.56**	5548 ± 1975
	SGD	88.26 ± 1.73	**3404±2654**
	SGD+M	83.01 ± 9.89	3877 ± 3269

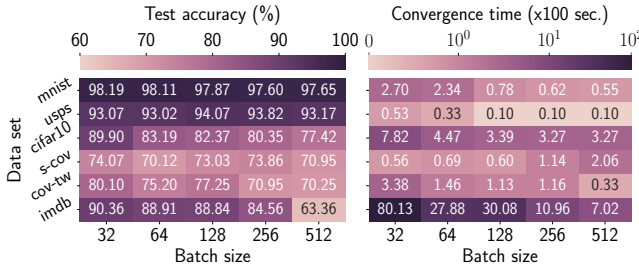


Figure 5: Kernel SGD performance under various batch sizes.

Kernel SGD. The results are illustrated in Figure 5. The decrease in batch size leads to a positive impact on the predictive accuracy on the tested data sets. This indicates that with less memory consumption for the kernel matrix, Kernel SGD can still achieve good predictive accuracy. When the batch size decreases, the convergence time increases for more vibrations in optimization with small batches. On the contrary, the convergence time for *s-cov* increases with the increasing batch size because of its small number of training instances. For a larger batch size, almost the whole *s-cov* data set which constitutes a batch is used to update the network only once. Therefore Kernel SGD needs more epochs and more time to converge in the training with *s-cov*.

5 RELATED WORK

We present the works that combine neural networks with kernel machines and introduce the effort made to improve the second-order optimization and preconditioned SGD in deep learning.

5.1 Neural Networks with Kernel Methods

Many studies take the advantages of both neural networks and kernel methods to better solve the machine learning problems. An arc-cosine kernel [5] was proposed to map the instances to the feature space which is similar to those produced by neural networks, but needs less computation. The experiments show that multi-layer kernel machines with arc-cosine kernels perform better than SVMs with RBF kernels and the deep belief net (DBN). Based on the observation that some match kernels over image patches can be regarded as variants of orientation histograms (i.e., low-level image features), Bo et al. [4] introduced three match kernels which represent the gradient, color and shape features of images to extract the patch-level features from the pixel features. Later, a convolutional multi-layer kernel [17] was proposed which is a generalization of kernel descriptors and a convolutional kernel network (CKN) was designed to approximately compute the proposed kernel. Mairal [16] improved the CKN with supervised learning and corresponding backpropagation procedures. Graph kernels [20] were proposed to measure the similarity between two graphs. Jacot et al. [12] demonstrated that the neural network problem can be solved following the gradient with respect to the proposed neural tangent kernel (NTK). NTK is the Kronecker product of the gradient of the realization function with respect to the parameters in ANN with itself. The computation of NTK needs to be repeated in training iterations while our kernel SGD uses fixed kernel matrix.

5.2 Second-Order Optimization

Due to the high computation cost, second-order optimizations have been extensively studied in solving the over-parameterized deep learning problems. The quasi-Newton method including L-BFGS [15], Gauss-Newton [26] and Kronecker-factored Approximate Curvature (K-FAC) [19] is a class of Newton methods which computes an approximate Hessian matrix. Xu et al. [30] proposes to compute the Hessian with sampled instances. Instead of computing the inexact Hessian matrix, Hessian-free methods [18] compute the Hessian-vector product in conjugate gradient without explicitly computing the Hessian matrix. Zhou et al. [33] uses both the full Hessian the semi-stochastic Hessian. ADAHESSIAN [32] integrates the first-order and second-order momentum in optimization where the second-order momentum is updated with the diagonal of Hessian. Spatial averaging is applied to the Hessian diagonal to mitigate the noise of Hessian. Nonetheless, the recomputation of derivatives is inevitable in these methods.

5.3 Preconditioned SGD

Preconditioned SGD transforms the gradient with a preconditioner to boost the optimization in ill-conditioned problems. Jacobi preconditioner is one of the most popular preconditioners and is improved by LeCun et al. [14] with Gauss-Newton matrix approximation. Gupta et al. [8] designed the "Shampoo" algorithm which generates a separate preconditioner for each dimension of a tensor. Some preconditioning methods exploit the second-order information as the preconditioner [24, 31]. Schaul et al. [24] use the Hessian matrix to dynamically update the learning rate for SGD. Dauphin et al. [6] proposed equilibrated SGD (ESGD) which applied an equilibration preconditioner to SGD. ESGD computes the inverse of the absolute Hessian matrix efficiently while keeping the ability to reduce the condition number and escape the saddle points. Although approximation techniques are used, the frequent update of preconditioners remains the main obstacle to making preconditioning practical.

6 CONCLUSION

To improve the second-order optimization in the neural network training, we have proposed Kernel SGD which exploits the second-order information from kernel machines. Kernel SGD prominently reduces the computation and memory cost during the training of neural networks with second-order optimization. We provided a theoretical convergence guarantee for the training using Kernel SGD. Our experimental results on tabular, image and text data have shown that Kernel SGD achieves an overall superior performance than other existing optimization methods, especially in generalization (e.g., 13% test accuracy improvement on the *imdb*) and convergence speed. Our findings may encourage more research on this direction of incorporating kernel methods with deep learning.

ACKNOWLEDGMENTS

This work was supported by the Natural Science Foundation of Guangdong Province, China (Grant No. 2022A1515010148), the National Natural Science Foundation of China (Grant No. 62177015 & No. 62072186), the Guangdong Basic and Applied Basic Research Foundation (Grant No. 2019B1515130001), the Guangzhou Science and Technology Planning Project (Grant No. 201904010197) and the Opening Project of Guangdong Key Laboratory of Big Data Analysis and Processing (No. 202002).

A PROOF OF THEOREM 3.1

Proof. Without loss of generality, we consider standard SGD that feeds one instance into the network in each iteration. Recall that the input instance $\boldsymbol{x}$ belongs to the i-th class and thus the cross entropy loss can be defined as $\mathcal{J}(W) = -y_i \ln a_i$, where a_i is the i-th output of the activation function. The loss $\mathcal{J}(W)$ is a function with respect to W and the weights of other layers can be treated as constants. As we use a softmax function to be the activation function where $a_i = e^{f_i}/\sum_{j=1}^{n_c} e^{f_j}$, the cross entropy loss can be rewritten in the from of Equation (9). Taking Equation (9) into the Inequality (10), we rewrite Inequality (10) as follows.

$$-f_i' + \ln \sum_{j}^{n_c} e^{f_j'} \leq -f_i + \ln \sum_{j}^{n_c} e^{f_j}.$$

where f_i' is the i-th input to the last layer with the updated weight W_i' and is derived as $f_i' = W_i' G(\boldsymbol{x})$. The weight W_i' is the i-th row of the updated weight W' which can be computed using $W_i' = W_i - \eta \nabla_{W_i} \mathcal{J}(W) H^{-1}$ according to Equation (8). With the expressions of f_i' and W_i', the Inequality (10) can be expanded as follows.

$$\eta \nabla_{W_i} \mathcal{J}(W) H^{-1} G(\boldsymbol{x}) + \ln \sum_{j}^{n_c} e^{f_j'} \leq \ln \sum_{j}^{n_c} e^{f_j}. \tag{12}$$

We take the natural exponential function on both sides and obtain

$$\sum_{j}^{n_c} \exp(f_j' + \eta \nabla_{W_i} \mathcal{J}(W) H^{-1} G(\boldsymbol{x})) \leq \sum_{j}^{n_c} \exp(f_j). \tag{13}$$

In Inequality (13), if each term in the summation on the left side is less than or equal to the corresponding term on the right side, then Inequality (13) holds. Thus we take the j-th term on each side and prove that for all j in $\{1, \ldots, n_c\}$, we have $E_j \leq 0$, where

$$E_j = \eta[\nabla_{W_i}\mathcal{J}(W) - \nabla_{W_j}\mathcal{J}(W)]H^{-1}G(\boldsymbol{x}). \tag{14}$$

Based on the definition of $\mathcal{J}(W)$, we can compute the gradient of loss $\mathcal{J}(W)$ with respect to W_j as follows.

$$\begin{aligned}\nabla_{W_j}\mathcal{J}(W) &= [\nabla_{W_{j1}}\mathcal{J}(W)\ \nabla_{W_{j2}}\mathcal{J}(W)\ \ldots\ \nabla_{W_{jd_h}}\mathcal{J}(W)] \\ &= (a_j - y_j)G(\boldsymbol{x})^T,\end{aligned} \tag{15}$$

where $\nabla_{W_j}\mathcal{J}(W) \in \mathbb{R}^{d_h}$ is a row vector. Substituting $\nabla_{W_j}\mathcal{J}(W)$ in Equation (14) with the result of Equation (15), we have

$$E_j = \eta(a_i - a_j - y_i + y_j)G(\boldsymbol{x})^T H^{-1}G(\boldsymbol{x}).$$

Since a_i is the value of a softmax function, we have that $0 \leq a_i \leq 1$ and can derive the following formulas.

$$a_i - a_j - y_i + y_j = \begin{cases} 0 & i = j, \\ a_i - a_j - 1 \leq 0 & i \neq j, \end{cases}$$

which can be integrated as $(a_i - a_j - y_i + y_j) \leq 0$. Since the learning rate η is greater than or equal to zero, we can derive that $\eta(a_i - a_j - y_i + y_j) \leq 0$ always holds. Then the last term in E_j to determine is $G(\boldsymbol{x})^T H^{-1}G(\boldsymbol{x})$. From the definition of positive semi-definite matrix, we can prove that H^{-1} and the reshaped H^{-1} are both positive semi-definite. Therefore, for any vector $G(\boldsymbol{x})$, we always have $G(\boldsymbol{x})^T H^{-1}G(\boldsymbol{x}) \geq 0$. Hence $E_j \leq 0$ is proved.

We summarize from the bottom up. As E_j is less than or equal to 0, Inequality (13) and Inequality (12) are satisfied. Hence we can prove that the loss decreases or stays unchanged as the training progresses when using Kernel SGD. □

B PROOF OF PROPOSITION 3.2

Proof. First, we compute the second derivative of the loss. Let F be the first derivative which is $\nabla_W\mathcal{J}(W) \in \mathbb{R}^{n_c \times d_h}$ and $\hat{F}$ denotes $\nabla_{\hat{W}}\hat{\mathcal{J}}(\hat{W}) \in \mathbb{R}^{n_c \times d_h}$. Then we represent the second derivative as $\nabla^2_W\mathcal{J}(W)$ where $\nabla^2_W\mathcal{J}(W) = \frac{\partial F}{\partial W}$. Based on the relation between the derivatives (i.e., $\frac{\partial F}{\partial W}$) and differentials (i.e., dF and dW), we have

$$\text{vec}(dF) = \frac{\partial F}{\partial W}^T \text{vec}(dW), \tag{16}$$

where $\text{vec}(W) = [W_{11}, \ldots, W_{n_c 1}, \ldots, W_{1d_h}, \ldots, W_{n_c d_h}]^T$. For differentials, we have

$$\begin{aligned} dF &= (d\nabla_{\hat{W}}\hat{\mathcal{J}}(\hat{W}))H^{\frac{1}{2}} + \nabla_{\hat{W}}\hat{\mathcal{J}}(\hat{W})(dH^{\frac{1}{2}}) = (d\hat{F})H^{\frac{1}{2}}, \\ dW &= (d\hat{W})H^{-\frac{1}{2}} + \hat{W}dH^{-\frac{1}{2}} = (d\hat{W})H^{-\frac{1}{2}}, \end{aligned}$$

where $dH^{\frac{1}{2}} = 0$ and $dH^{-\frac{1}{2}} = 0$. Suppose H is a d_h-by-d_h matrix where m is equal to d_h. Then we we have the vectorization of differentials dF and dW as follows.

$$\text{vec}(dF) = \text{vec}((d\hat{F})H^{\frac{1}{2}}) = (H^{\frac{1}{2}} \otimes I)\text{vec}(d\hat{F}), \tag{17}$$

$$\text{vec}(dW) = \text{vec}(d(\hat{W})H^{-\frac{1}{2}}) = (H^{-\frac{1}{2}} \otimes I)\text{vec}(d\hat{W}), \tag{18}$$

where $\otimes$ is the Kronecker product and I is an $n_c \times n_c$ identity matrix. Using the definition in Equation (17) and Equation (18), we can rewrite Equation (16) as follows.

$$\begin{aligned} &(H^{\frac{1}{2}} \otimes I)\text{vec}(d\hat{F}) = \frac{\partial F}{\partial W}^T (H^{-\frac{1}{2}} \otimes I)\text{vec}(d\hat{W}). \\ \Rightarrow\ &\text{vec}(d\hat{F}) = (H^{-\frac{1}{2}} \otimes I)\frac{\partial F}{\partial W}^T (H^{-\frac{1}{2}} \otimes I)\text{vec}(d\hat{W}). \end{aligned} \tag{19}$$

Since we already have that $\text{vec}(d\hat{F}) = \frac{\partial \hat{F}}{\partial \hat{W}}^T \text{vec}(d\hat{W})$, combining with Equation (19) and we have

$$\frac{\partial \hat{F}}{\partial \hat{W}} = (H^{-\frac{1}{2}} \otimes I)\frac{\partial F}{\partial W}(H^{-\frac{1}{2}} \otimes I). \tag{20}$$

Equation (20) shows the relation between the second derivative of the transformed loss and second derivative of the original loss.

Then we compute the first-order Taylor expansion of the projected loss near the point $\hat{W}_0$ and the original loss near the point W_0, respectively. We take derivatives on both sides of the expanded equations. Since $\hat{W}^*$ and W^* are the minimums of the projected and original loss, respectively, we derive that

$$\text{vec}(\Delta\hat{W}) = \text{vec}(\hat{W}^* - \hat{W}_0) = -\nabla^2_{\hat{W}}\hat{\mathcal{J}}(\hat{W}_0)^{-1}\text{vec}(\nabla_{\hat{W}}\hat{\mathcal{J}}(\hat{W}_0)), \tag{21}$$

$$\text{vec}(\Delta W) = \text{vec}(W^* - W_0) = -\nabla^2_W\mathcal{J}(W_0)^{-1}\text{vec}(\nabla_W\mathcal{J}(W_0)). \tag{22}$$

According to the definition in Equation (20), we can rewrite Equation (21) below.

$$\begin{aligned} \text{vec}(\Delta\hat{W}) &= -(\frac{\partial \hat{F}_0}{\partial \hat{W}})^{-1}\text{vec}(\nabla_{\hat{W}}\hat{\mathcal{J}}(\hat{W}_0)) \\ &= -(H^{\frac{1}{2}} \otimes I)\nabla^2_W\mathcal{J}(W_0)^{-1}(H^{-\frac{1}{2}} \otimes I)^{-1}\text{vec}(\nabla_{\hat{W}}\hat{\mathcal{J}}(\hat{W}_0)), \end{aligned} \tag{23}$$

where $\frac{\partial \hat{F}_0}{\partial \hat{W}} = \nabla^2_{\hat{W}}\hat{\mathcal{J}}(\hat{W}_0)$ and $\frac{\partial F_0}{\partial W} = \nabla^2_W\mathcal{J}(W_0)$. With the fact that $\text{vec}(\nabla_{\hat{W}}\hat{\mathcal{J}}(\hat{W}_0)) = (H^{-\frac{1}{2}} \otimes I)\text{vec}(\nabla_W\mathcal{J}(W_0))$, Equation (23) can be written as follows.

$$\text{vec}(\Delta\hat{W}) = (H^{\frac{1}{2}} \otimes I)\text{vec}(\Delta W). \tag{24}$$

Then we take Euclidean norm $||\cdot||_2$ on both sides and obtain

$$\begin{aligned} ||\text{vec}(\Delta\hat{W})||_2 &= ||(H^{\frac{1}{2}} \otimes I)\text{vec}(\Delta W)||_2 \\ &\leq ||H^{\frac{1}{2}} \otimes I||_F||\text{vec}(\Delta W)||_2 \\ &= \sqrt{n_c}\ ||H^{\frac{1}{2}}||_F||\text{vec}(\Delta W)||_2, \end{aligned} \tag{25}$$

where $||\cdot||_F$ is the Frobenius norm. The inequality in Formula (25) is derived from the property that the Frobenius norm of a matrix is compatible with the Euclidean norm of a vector (i.e., $||A\boldsymbol{v}||_2 \leq ||A||_F||\boldsymbol{v}||_2$ where $A \in \mathbb{R}^{n \times n}$ and $\boldsymbol{v} \in \mathbb{R}^n$). We use eigen-decomposition on H to derive that $H^{\frac{1}{2}} = Q\Lambda^{\frac{1}{2}}Q^T$ where Λ is the eigenvalue matrix of H. With the last result of Formula (25), we have

$$\begin{aligned} \left\|\text{vec}(\Delta\hat{W})\right\|_2 &\leq \sqrt{n_c}\left\|Q\Lambda^{\frac{1}{2}}Q^T\right\|_F\left\|\text{vec}(\Delta W)\right\|_2 \\ &= \sqrt{n_c}\left\|\Lambda^{\frac{1}{2}}\right\|_F\left\|\text{vec}(\Delta W)\right\|_2 \\ &= \sqrt{n_c \cdot \sum_{i=1}^{m}(\pi_i^{\frac{1}{2}})^2}\left\|\text{vec}(\Delta W)\right\|_2, \end{aligned} \tag{26}$$

where π_i is the i-th eigenvalue of matrix H. If the assumption is satisfied where $\sum_{i=1}^{m}\pi_i \leq \frac{1}{n_c}$, we have $||\text{vec}(\hat{W}^* - \hat{W})||_2 \leq ||\text{vec}(W^* - W)||_2$ (i.e., $||\hat{W}^* - \hat{W}||_F \leq ||W^* - W||_F$). If m is not equal to d_h, we compute the pseudo inverse of $H^{-\frac{1}{2}}$ and can obtain the same conclusion with similar process of proof. □

REFERENCES

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

[2] Albert S. Berahas, Jorge Nocedal, and Martin Takáč. 2016. A Multi-Batch L-BFGS Method for Machine Learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain). 1063–1071.

[3] Christopher M Bishop and Nasser M Nasrabadi. 2006. *Pattern recognition and machine learning*. Vol. 4. Springer.

[4] Liefeng Bo, Xiaofeng Ren, and Dieter Fox. 2010. Kernel descriptors for visual recognition. In *Advances in neural information processing systems (NeurIPS)*. 244–252.

[5] Youngmin Cho and Lawrence K Saul. 2009. Kernel methods for deep learning. In *Advances in neural information processing systems (NeurIPS)*. 342–350.

[6] Yann Dauphin, Harm De Vries, and Yoshua Bengio. 2015. Equilibrated adaptive learning rates for non-convex optimization. In *Advances in neural information processing systems (NeurIPS)*. 1504–1512.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).

[8] Vineet Gupta, Tomer Koren, and Yoram Singer. 2018. Shampoo: Preconditioned stochastic tensor optimization. In *International conference on machine learning (ICML)*. PMLR, 1842–1850.

[9] Yilong Hao, Kanishka Tyagi, Rohit Rawat, and Michael Manry. 2016. Second order design of multiclass kernel machines. In *2016 International joint conference on neural networks (IJCNN)*. IEEE, 3233–3240.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*. 770–778.

[11] Thomas Hofmann, Bernhard Schölkopf, and Alexander J Smola. 2008. Kernel methods in machine learning. *The annals of statistics* 36, 3 (2008), 1171–1220.

[12] Arthur Jacot, Franck Gabriel, and Clément Hongler. 2018. Neural tangent kernel: Convergence and generalization in neural networks. *arXiv preprint arXiv:1806.07572* (2018).

[13] S Sathiya Keerthi and Chih-Jen Lin. 2003. Asymptotic behaviors of support vector machines with Gaussian kernel. *Neural computation* 15, 7 (2003), 1667–1689.

[14] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. 2012. Efficient backprop. In *Neural networks: Tricks of the trade*. Springer, 9–48.

[15] Dong C Liu and Jorge Nocedal. 1989. On the limited memory BFGS method for large scale optimization. *Mathematical programming* 45, 1-3 (1989), 503–528.

[16] Julien Mairal. 2016. End-to-end kernel learning with supervised convolutional kernel networks. In *Advances in neural information processing systems (NeurIPS)*. 1399–1407.

[17] Julien Mairal, Piotr Koniusz, Zaid Harchaoui, and Cordelia Schmid. 2014. Convolutional kernel networks. In *Advances in neural information processing systems (NeurIPS)*. 2627–2635.

[18] James Martens. 2010. Deep learning via hessian-free optimization.. In *International conference on machine learning (ICML)*, Vol. 27. 735–742.

[19] James Martens and Roger Grosse. 2015. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning (ICML)*. 2408–2417.

[20] Nicolò Navarin, Dinh V Tran, and Alessandro Sperduti. 2018. Pre-training graph neural networks with kernels. *arXiv preprint arXiv:1811.06930* (2018).

[21] Dat Quoc Nguyen, Thanh Vu, Afshin Rahimi, Mai Hoang Dao, Linh The Nguyen, and Long Doan. 2020. WNUT-2020 Task 2: Identification of Informative COVID-19 English Tweets. In *Proceedings of the 6th workshop on noisy user-generated text (W-NUT)*. 314–318. https://www.aclweb.org/anthology/2020.wnut-1.41

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems (NeurIPS)*. 8026–8037. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[23] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical methods in natural language processing (EMNLP)*. 1532–1543. http://www.aclweb.org/anthology/D14-1162

[24] Tom Schaul, Sixin Zhang, and Yann LeCun. 2013. No more pesky learning rates. In *International conference on machine learning (ICML)*. 343–351.

[25] Bernhard Schölkopf, Ralf Herbrich, and Alex J Smola. 2001. A generalized representer theorem. In *International conference on computational learning theory*. Springer, 416–426.

[26] Nicol N Schraudolph. 2002. Fast curvature matrix-vector products for second-order gradient descent. *Neural computation* 14, 7 (2002), 1723–1738.

[27] Alex J Smola and Bernhard Schölkopf. 1998. *Learning with kernels*. Vol. 4. Citeseer.

[28] Eduardo Soares, Plamen Angelov, Sarah Biaso, Michele Higa Froes, and Daniel Kanda Abe. 2020. SARS-CoV-2 CT-scan dataset: A large dataset of real patients CT scans for SARS-CoV-2 identification. *medRxiv* (2020). https://doi.org/10.1101/2020.04.24.20078584

[29] Zeyi Wen, Zhishang Zhou, Hanfeng Liu, Bingsheng He, Xia Li, and Jian Chen. 2021. Enhancing SVMs with Problem Context Aware Pipeline. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 1821–1829.

[30] Peng Xu, Fred Roosta, and Michael W Mahoney. 2020. Newton-type methods for non-convex optimization under inexact hessian information. *Mathematical Programming* 184, 1 (2020), 35–70.

[31] Zhewei Yao, Amir Gholami, Daiyaan Arfeen, Richard Liaw, Joseph Gonzalez, Kurt Keutzer, and Michael Mahoney. 2018. Large batch size training of neural networks with adversarial training and second-order information. *arXiv preprint arXiv:1810.01021* (2018).

[32] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael Mahoney. 2021. ADAHESSIAN: An Adaptive Second Order Optimizer for Machine Learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 12 (2021), 10665–10673.

[33] Dongruo Zhou, Pan Xu, and Quanquan Gu. 2019. Stochastic Variance-Reduced Cubic Regularization Methods. *J. Mach. Learn. Res.* 20, 134 (2019), 1–47.