Data-driven algorithm selection and tuning in optimization and signal processing

Jesús A. De Loera, Jamie Haddock, Anna Ma & Deanna Needell

Annals of Mathematics and Artificial Intelligence

ISSN 1012-2443

Ann Math Artif Intell
DOI 10.1007/s10472-020-09717-z

Springer

Annals of Mathematics and Artificial Intelligence
https://doi.org/10.1007/s10472-020-09717-z

Data-driven algorithm selection and tuning in optimization and signal processing

Jesús A. De Loera[1] · Jamie Haddock[2] · Anna Ma[3] · Deanna Needell[2]

Accepted: 27 October 2020Published online: 12 November 2020

Abstract
Machine learning algorithms typically rely on optimization subroutines and are well known to provide very effective outcomes for many types of problems. Here, we flip the reliance and ask the reverse question: can machine learning algorithms lead to more effective outcomes for optimization problems? Our goal is to train machine learning methods to automatically improve the performance of optimization and signal processing algorithms. As a proof of concept, we use our approach to improve two popular data processing subroutines in data science: stochastic gradient descent and greedy methods in compressed sensing. We provide experimental results that demonstrate the answer is "yes", machine learning algorithms do lead to more effective outcomes for optimization problems, and show the future potential for this research direction. In addition to our experimental work, we prove relevant *Probably Approximately Correct* (PAC) learning theorems for our problems of interest. More precisely, we show that there exists a learning algorithm that, with high probability, will select the algorithm that optimizes the average performance on an input set of problem instances with a given distribution.

Keywords Automated machine learning · Compressed sensing · Neural networks · Algorithm selection · Hyperparameter tuning

Mathematics Subject Classification (2010) 65K10 · 90C26 · 68T05 · 68T07

✉ Jamie Haddock
jhaddock@math.ucla.edu

Jesús A. De Loera
deloera@math.ucddavis.edu

Anna Ma
anna.ma@uci.edu

Deanna Needell
deanna@math.ucla.edu

[1] University of California, Davis, CA, USA

[2] University of California, Los Angeles, CA, USA

[3] University of California, Irvine, CA, USA

1 Introduction

Machine learning is a popular and powerful tool that has emerged at the forefront of a vast array of applications (most famously in image processing). At their core, neural nets rely on solving nonlinear optimization problems. From this point of view, improving key optimization subroutines and other auxiliary data processing methods directly helps to improve learning methods. Here, we aim to use basic machine learning algorithms to improve optimization and signal processing subroutines by choosing the best algorithm or the best choice of parameters in concrete instances.

Practitioners of optimization and signal processing know that for a given input instance, there are choices among several algorithms or internal parameters that require fine-tuning in order to obtain the best performance (e.g., should one use the simplex method or interior-point methods?). These choices often lead to drastically different performances and thus such decisions are crucial in many applications. The questions we consider here are: *What is the best algorithm to use given a new data instance? What is the best choice of parameters for a given data instance?* Often humans can use their expertise and experience to narrow down algorithm or parameter choices, but what can be done by the nonexpert user to make sound decisions? In our approach, which we refer to as the *empirical algorithm selection and tuning* approach, we use available *individual problem instances* as data to train a neural network that can assist in the tuning of parameters or in algorithm selection. This approach does not require a human expert to reduce the algorithm or parameter search space and can therefore be utilized by nonexperts during the optimization process. We illustrate our approach with stochastic gradient descent and greedy methods in compressed sensing.

1.1 Our contributions

In this work we study the problems of algorithm selection in *compressed sensing* and of parameter tuning for *Stochastic Gradient Descent* (SGD) algorithms. In both cases we wish to train a recommendation or classification algorithm that can output an optimal algorithmic decision (method, parameters, etc.) for a given input data set. Note that in contrast to previous work in step size tuning for SGD, our work does not require that iteration specific step sizes be computed at every step. We mostly study these problems from the experimental point of view, but we also prove PAC (probably approximately correct) learning theorems for our problems of interest. Our main contributions are as follows.

- In Section 2 we apply our methodology, through concrete experiments, to the selection of compressed sensing algorithms. Here we concentrate on selecting the best among three well-known greedy algorithms for solving the compressed sensing problem: *Hard Thresholding Pursuit* (HTP) [24], *Normalized Iterative Hard Thresholding* (NIHT) [12], and *Compressive Sampling Matching Pursuit with Subspace Pursuit* (CSMPSP) [43, 46]. We have been inspired by the work of [11], where the authors catalog optimal algorithm selection through brute force experimental testing. Although our machine learning approach is useful precisely when such a rigorous catalog is *not* available, the work of [11] will be used as validation of our framework.
- In Section 3 we apply our methodology, through concrete experiments, to the selection of the best step size in the popular *stochastic gradient descent algorithm* [49], which itself is used as a subroutine in many learning frameworks. Unfortunately, tuning the

step size (also called the *learning rate*) is often more an art than a science, and the selection can lead to drastically different overall behavior. We aim to alleviate this issue by allowing for such selections to be done by the trained machine.

– In Section 5 we prove two PAC theorems about our SGD and compressed sensing applications. In particular, we prove that there exists a simple learning algorithm which, with high probability, selects a learning rate, from among an interval of learning rates, that achieves nearly the optimal average SGD performance for a given distribution of functions. Additionally, we point out that there is a simple learning algorithm which, with high probability, selects the compressed sensing algorithm from the finite set of considered greedy algorithms which achieves the optimal average performance for a given distribution of functions.

For our experiments we use Neural Networks for classification. Neural Networks are computing systems inspired by the biological neural networks. They have shown remarkable success in various machine learning tasks including classification [36]. While there are plenty of sophisticated, state of the art neural net architectures such as GoogLeNet [54], ResNet [30], DenseNet [32] and CliqueNet [59], we will demonstrate that even simple networks that do not have to be run on expensive remote processors can aid in algorithm selection. This neural net learning approach is perfect for learning and modeling nonlinear and complex relationships allowing, as the name suggests, the machine to learn data relationships by itself.

1.2 Prior relevant work

One can trace the first interactions between machine learning and optimization at least to the early 1990's [53] and it is an interaction that continues to grow. The problem of interest is that of efficiently automating the selection of algorithms or their parameter configurations. It has attracted attention by many authors, and relies on multiple techniques. Within Artificial Intelligence it has received several names: *algorithm selection*, *algorithm configuration*, *self-adapting algorithms*, or simply *automated machine learning (autoML)*. We do not attempt to be comprehensive on our survey of this vast body of literature. Instead, we refer the reader to [60] for a more complete review on autoML and focus here on works that are more relevant to the proceeding discussion.

Let us first mention that the list of tools and techniques used is very diverse. Naive approaches to autoML began with simple grid searches over hyperparameter spaces. In the work of [8], the authors propose random searches over the hyperparameter space. However, these approaches do not incorporate any learning and each instance is independent of prior knowledge or computation. In more recent years, researchers have been proposing more sophisticated techniques. For example, in [38] the problem is approached as a Markov Decision problem. In [58] the problem is approached with techniques similar to the so-called matrix completion method (see e.g. [14, 15]). A theoretical learning framework for algorithm selection was presented in [28] with a follow up in [4]. Hyperparameter optimization was formulated and solved as an infinite-armed bandit problem in [41]. Some authors have used an empirical hardness model to predict the running time of algorithms and have applied this approach to improve Satisfiability (SAT) solvers [21, 40]. While many of the aforementioned approaches and many others [5, 6, 23, 48] require a preprocessing step before algorithm or parameter selection can be performed, our approach simply uses the data that encodes the problem (or even simpler attributes of the data) as input features to our learning approach.

Several authors have been directly concerned with algorithm selection and tuning for optimization (see e.g., [2, 3, 34] and the many references therein). Recently a survey of ways to use learning in combinatorial optimization was presented in [7]. In [9] the authors redefine mixed integer convex optimization problems as a multiclass classification problem where the machine learning predictor gives insights on the optimal solution. Optimization, especially combinatorial optimization, relies on efficient heuristics. For example, Dai et al. [33] develop a method to learn heuristics over graph problems. As another example, branch-and-bound involves decisions about the branching variable. Users struggle to select the best rule to perform this heuristic and several authors have proposed ways to use machine learning to select the best branching rules (see [1, 34]). Machine learning methods have also been useful in aiding the selection of reformulations and decompositions for mixed-integer optimization. For example, [13, 37] used machine learning to decide which instances more efficiently solve mixed-integer quadratic optimization problems. Note that many of these works are combinatorial in nature. In comparison, our work demonstrates that our approach can drive optimal selection even in continuous and analytical problems.

Most related to our work is the recent work of [31] where the authors use convolutional neural nets (CNN) as a black box for algorithm selection. Their approach involves a two step process. First, they capture a literal image of the landscape of the optimization problem and then feed that image into a CNN to classify optimal algorithms for a given data instance. In comparison to our approach, we do not require the intermediate step of concretely capturing the landscape of the optimization problem at hand. We instead feed the input data directly into the neural network, creating an even more rudimentary approach than what is proposed in [31]. Despite its simplicity, our approach provides useful and promising results.

While there has been much work in the area of autoML and automated algorithm selection, these automated approaches differ from ours in terms of their application to new data. Our straightforward approach allows nonexpert practitioners to apply this algorithm selection approach without the expertise required to preprocess the data or determine meta-features that enable an effective learning approach. In addition, we demonstrate that such simplistic approaches not only yield advantageous results, but can also be used for continuous and analytical problems.

1.3 Notation

Here and throughout the paper, we write

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{y}, \tag{1}$$

where $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ is the measurement (or data) matrix, $\boldsymbol{y} \in \mathbb{R}^m$ is the measurement vector, and $\boldsymbol{x} \in \mathbb{R}^n$ is the signal being recovered. We use $(\cdot)^*$ to denote the transpose operator and $(\cdot)^\dagger$ denotes the pseudoinverse.

In the compressed sensing problem considered in Section 2, the measurement matrix is underdetermined ($m \ll n$) and the signal is assumed to be sparse; in particular, we say $\boldsymbol{x}$ is s-sparse when it has at most s nonzero entries. Furthermore, for any vector $\boldsymbol{v}$, $\text{supp}_s(\boldsymbol{v})$ returns the indices corresponding to the s largest in magnitude entries of $\boldsymbol{v}$ and $P_T(\boldsymbol{v})$ returns a vector whose entries are 0 outside of the support of set T and equal to $\boldsymbol{v}$ on the support of T. For any set $T \subset \{1, \cdots, n\}$, a matrix $\boldsymbol{A}$ constrained to the columns indexed by T is denoted by $\boldsymbol{A}_T \in \mathbb{R}^{m\times|T|}$.

In the least-squares problems considered in Section 3, the measurement matrix is overdetermined ($m \gg n$) and no sparsity assumption is made on the signal $\boldsymbol{x}$. We use the recovery

error and the residual error at the M-th iteration of SGD, $\|\boldsymbol{x}_M - \boldsymbol{x}\|_2$ and $\|\boldsymbol{A}\boldsymbol{x}_M - \boldsymbol{y}\|_2$ respectively, to measure the performance of the algorithm with given learning rates.

In both Sections 2 and 3, we will make use of a maximum iteration cap to stop the algorithm in question, which we denote M. The use of this stopping criterion is described in detail in each section.

2 Application I: compressed sensing algorithm selection

We begin the investigation of our framework with a proof of concept inspired by the work done in [11] where the authors rigorously test various compressed sensing methods under various settings in a brute-force way. We will show that we can use neural networks to recover the phase transitions that were acquired via rigorous testing in the aforementioned paper. For this reason, we adopt a similar algorithmic and experimental setup. First, we will explain the compressed sensing problem and notation used throughout, then we will present three greedy algorithms. Following that, the experimental setup including the different sensing matrices, signal initialization, and stopping criteria are discussed. Finally, we present our experimental results and remark on our findings.

There is now an abundance of both theory and algorithms that guarantee robust and accurate recovery of sparse signals, under various assumptions on the measurement matrix $\boldsymbol{A}$ [22, 25]. For example, the so-called *Restricted Isometry Property* [16] guarantees such recovery and random matrix constructions are shown to satisfy this property when the number of measurements m scales like $s \log n$ [50]. Under this or related assumptions, both greedy (iterative) algorithms and optimization-based methods (e.g., L1-minimization) are shown to produce accurate recovery results. In general, the performance of such algorithms depends on the undersampling and oversampling rates which we denote as

$$\delta = \frac{m}{n} \text{ and } \rho = \frac{s}{m}, \tag{2}$$

respectively. Furthermore, we refer to combinations of δ and ρ as the *(δ, ρ) plane*. By observing the behavior of algorithms on the (δ, ρ) plane, we can see how different approaches act under various sampling rates.

We consider three greedy algorithms for solving the compressed sensing problem: *Hard Thresholding Pursuit* (HTP) [24], *Normalized Iterative Hard Thresholding* (NIHT) [12], and *Compressive Sampling Matching Pursuit with Subspace Pursuit* (CSMPSP) [43, 46]. The pseudocode for HTP, NIHT, and CSMPSP appears in Algorithms 1, 2, and 3 respectively. These methods are all similar in spirit; they seek to recover the signal $\boldsymbol{x}$ from $\boldsymbol{y}$ while also identifying the support of $\boldsymbol{x}$, which is discovered iteratively. Each essentially uses a *proxy* for the signal $\boldsymbol{x}$ (e.g., $\boldsymbol{A}^*\boldsymbol{y}$) to identify a support estimate T, then estimates $\boldsymbol{x}$ on that support (e.g., $\boldsymbol{x}_t = \boldsymbol{A}_T^{\dagger}\boldsymbol{y}$) , then computes the residual and repeats the process to locate the remainder of $\boldsymbol{x}$. HTP and NIHT use specially chosen step sizes (denoted w_k) when updating the estimate to $\boldsymbol{x}$ and recompute the support in each iteration, whereas CSMPSP uses a union of prior estimates followed by pruning. See Algorithms 1, 2 and 3 and [12, 24, 43] for details about these approaches. What is important for our purpose is that each algorithm may perform differently for a given set of inputs, leading to varying accuracy on the output. Therefore, there is value in using machine learning tools to decide what is the best choice of algorithm in a given problem instance. Also note that each algorithm

takes the same inputs, namely the measurement matrix $\boldsymbol{A}$, the measurement vector $\boldsymbol{y}$, and an approximation for the number of nonzero entries s in the sparse signal.

Although the theory for these approaches holds *uniformly*, meaning it holds for any sparse signal and matrix satisfying the assumptions, it has long been observed that the algorithms actually behave quite differently on various kinds of signal and measurement ensembles [11, 17, 26]. In fact, [11] documents an extensive comparison of these approaches for various ensembles while ranging the parameters δ and ρ. This latter work can be used as a "lookup table," when one knows the input information and wants to select the optimal algorithm for their purpose. Their work, in some sense, motivates us to apply the machine learning methodology to compressed sensing, as we have a comprehensive benchmark with which to compare these methods. Note that these comparisons were made in a brute force manner, where each method was run on each ensemble type over a fine grid of input parameters. Such an exhaustive approach is not practical when the input domain is extremely large. Moreover, in this setting, we have a greater understanding of how these greedy algorithms will behave for a specific problem instances, making it an appropriate problem to verify and validate our framework.

Algorithm 1 Hard thresholding pursuit.

procedure HTP($\boldsymbol{A}$, $\boldsymbol{y}$, s)
  Initialize $\boldsymbol{x}_0 = \boldsymbol{A}^*\boldsymbol{y}$, $T_0 = \text{supp}_s(\boldsymbol{x}_0)$, $\boldsymbol{x}_0 = P_{T_0}(\boldsymbol{x}_0)$, $\boldsymbol{r}_0 = \boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}_0$, $k = 1$
  while stopping criteria not reached **do**
    $w_k = \frac{\|(\boldsymbol{A}^*\boldsymbol{r}_{k-1})_{T_{k-1}}\|_2^2}{\|\boldsymbol{A}_{T_{k-1}}(\boldsymbol{A}^*\boldsymbol{r}_{k-1})_{T_{k-1}}\|_2^2}$
    $\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \boldsymbol{w}_k\boldsymbol{A}^*\boldsymbol{r}_{k-1}$
    $T_k = \text{supp}_s(\boldsymbol{x}_k)$
    $\boldsymbol{x}_k = \boldsymbol{A}_{T_k}^{\dagger}\text{y}$
    $\boldsymbol{r}_k = \boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}_{k-1}$
    $k = k + 1$
  end while
end procedure

Algorithm 2 Normalized iterative hard thresholding.

procedure NIHT($\boldsymbol{A}$, $\boldsymbol{y}$, s)
  Initialize $\boldsymbol{x}_0 = \boldsymbol{A}^*\boldsymbol{y}$, $T_0 = \text{supp}_s(\boldsymbol{x}_0)$, $\boldsymbol{x}_0 = P_{T_0}(\boldsymbol{x}_0)$, $\boldsymbol{r}_0 = \boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}_0$, $k = 1$
  while stopping criteria not reached **do**
    $w_k = \frac{\|(\boldsymbol{A}^*\boldsymbol{r}_{k-1})_{T_{k-1}}\|_2^2}{\|\boldsymbol{A}_{T_{k-1}}(\boldsymbol{A}^*\boldsymbol{r}_{k-1})_{T_{k-1}}\|_2^2}$
    $\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \boldsymbol{w}_k\boldsymbol{A}^*\boldsymbol{r}_{k-1}$
    $T_k = \text{supp}_s(\boldsymbol{x}_k)$
    $\boldsymbol{x}_k = P_{T_k}(\boldsymbol{x}_k)$
    $\boldsymbol{r}_k = \boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}_k$
    $k = k + 1$
  end while
end procedure

Algorithm 3 Compressive sampling matching pursuit with subspace pursuit.

procedure CSMPSP($\boldsymbol{A}$, $\boldsymbol{y}$, s)
  Initialize $\boldsymbol{x}_0 = \boldsymbol{A}^*\boldsymbol{y}$, $T_0 = \text{supp}_k(\boldsymbol{x}_0)$, $\boldsymbol{x}_0 = P_{T_0}(\boldsymbol{x}_0)$, $\boldsymbol{r}_0 = \boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}_0$, $k = 1$
  while stopping criteria not reached **do**
    $S_k = \text{supp}(\boldsymbol{A}^*\boldsymbol{r}_{k-1})$
    $\Lambda_k = T_{k-1} \cup S_k$
    $\boldsymbol{x}_k = \boldsymbol{A}_{\Lambda_k}^{\dagger}\boldsymbol{y}$
    $T_k = \text{supp}_s(\boldsymbol{x}_k)$
    $\boldsymbol{x}_k = P_{T_k}(\boldsymbol{x}_k)$
    $\boldsymbol{r}_k = \boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}_k$
    $k = k + 1$
  end while
end procedure

2.1 Experimental setup

We consider three randomly generated measurement matrices for this setting: Gaussian, Sparse, and Discrete Cosine Transform (DCT). Entries of the Gaussian matrices are drawn $i.i.d.$ from $\mathcal{N}(0, \frac{1}{m})$ so that in expectation, they have normalized columns. Sparse measurement matrices have $p = 7$ nonzero entries in each column where the value of the nonzero entries is drawn from $\{\pm p^{-\frac{1}{2}}\}$ with equal probability. Finally the DCT measurement matrices consist of m randomly subsampled rows of the $n \times n$ full DCT matrix. The number of measurements m is determined by δ and the vector $\boldsymbol{x}$ being recovered has s nonzero entries (determined by ρ) and takes on values $\{\pm 1\}$ with equal probability where δ and ρ are as defined in (2). The measurement vector $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ where $\boldsymbol{A}$ is one of the three types of measurement matrices and $\boldsymbol{x}$ is the signal to be recovered.

We terminate any algorithm when it satisfies one of the following stopping criteria.

1. **Convergence** - An algorithm is convergent if the residual error is small enough. In particular, if
$$\|\boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{y}\| \leq 0.001\delta.$$
2. **Divergence** - An algorithm is divergent if the residual error is larger than a factor of the norm of the initial residual:
$$\|\boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{y}\| \geq 100\|\boldsymbol{A}\boldsymbol{x}_0 - \boldsymbol{y}\|.$$
3. **Slow Progress I** - After 750 iterations of NIHT or 150 iterations of CSMPSP or HTP, we begin to check for slow progress. For the first version of "slow progress" we check whether the residual has made any significant progress over the last 15 iterations:
$$\max_i |\|\boldsymbol{A}\boldsymbol{x}_{k-i+1} - \boldsymbol{y}\| - \|\boldsymbol{A}\boldsymbol{x}_{k-i} - \boldsymbol{y}\|| \leq 10^{-6}.$$
4. **Slow Progress II** - After 750 iterations of NIHT or 150 iterations of CSMPSP or HTP, we check whether the convergence rate is close to 1:
$$\left(\frac{\|\boldsymbol{A}\boldsymbol{x}_{k-15} - \boldsymbol{y}\|}{\|\boldsymbol{A}\boldsymbol{x}_k - \boldsymbol{y}\|}\right)^{\frac{1}{15}} \leq 0.999.$$
5. **Maximum Iteration** - An algorithm that runs for longer than 60 minutes (discounting time for computing metrics) or M iterations (where $M = 900$ for NIHT and $M = 200$ for CSMPSP and HTP) has reached the allowable computation time and is terminated.

It should be noted the algorithm stopping criteria of (1)–(4) are as in [11] while the last exit was added to keep from a single experiment from running for too long. Practically, the last stopping criteria reflects a computational time constraint.

2.2 Experiments

In the following set of experiments, we train neural networks to classify whether or not an algorithm can recover a signal in the standard compressed sensing problem (1). The experiment requires three phases: creating training data, training the neural network, and testing the neural network.

In the first phase, training data with labels are created to input into the neural network. The training data set comprises of 2241 samples. For each matrix type (Gaussian, Sparse, DCT), there are 747 training points on the (δ, ρ) plane (See (2)). For each (δ, ρ) pair, we run Algorithms 1, 2, and 3 until the algorithm satisfies one of the stopping criteria discussed in Section 2.1. In order for a given algorithm to be labeled as "successful" at recovering signals for a specified (δ, ρ) and measurement matrix, 50 of the 100 randomly generated samples must have satisfied the "convergence" stopping criteria. This phase is completed in MATLAB using version R2014b on a desktop running Linux.

The training data from the first phase and labels are used to train neural networks in the second phase. The input variables used by the neural network are the signal dimension n, the number of measurements m, the number of nonzero entries s, and an indicator variable that indicates the measurement matrix. The second phase is accomplished using Python 3

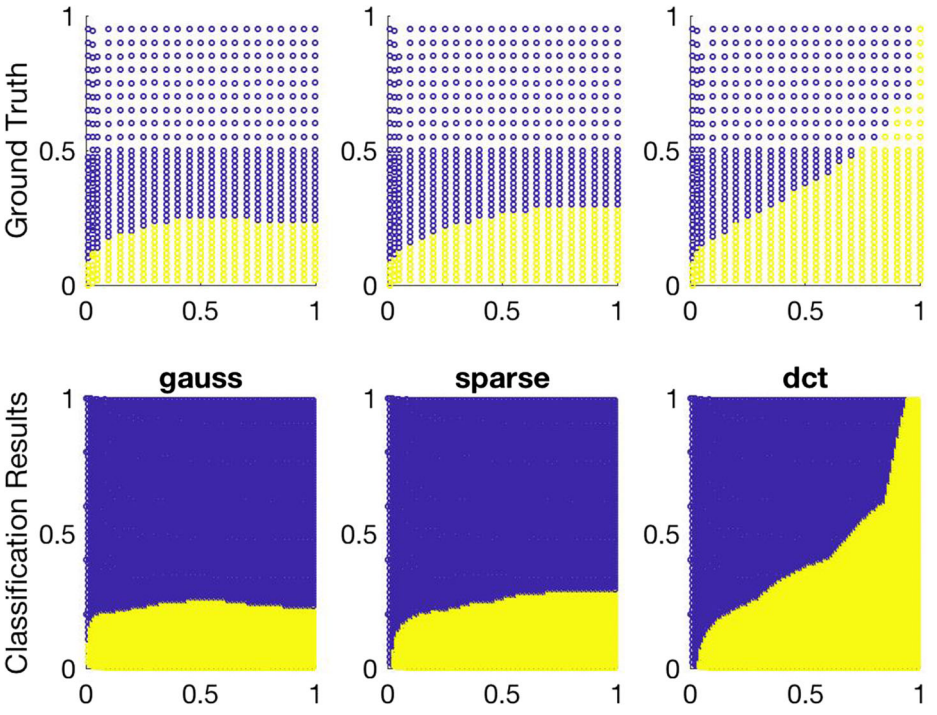

Fig. 1 Recovery for phase transitions of various types of measurement matrices for HTP. (Test validation accuracy = 0.969)

and Keras 2.1 with TensorFlow as a back end. We set up the neural network to contain two hidden layers, the first with three nodes and a second layer with nine nodes, and offer the following intuition for the neural network structure. The purpose of the first layer is to determine the measurement matrix type while the second layer classifies whether or not an algorithm will be successful. The hidden layers utilize ReLu as their activation function, with the exception of the final output layer which uses the sigmoid function. Approximately 90% of the available data is used to train our neural network for each algorithm and the remaining 10% is used to measure validation accuracy on the trained network.

Figures 1, 2 and 3 present the computational results for HTP, NIHT, and CSMPSP respectively. In each subplot, the horizontal axis represents the value of δ and the vertical axis represents the value of ρ. Furthermore, each figure can be broken down as follows. Each column isolates a specific measurement matrix: Gaussian, sparse, and DCT (left to right). The first row of each figure shows the training (δ, ρ) pairs (i.e., data created in first phase of experiment) along with their labels, indicated by the color of the data point. Here, yellow points indicate that an algorithm is "successful" and blue points indicate that the algorithm is not successful. In the second row of each figure, we show results produced by the trained neural network from the second phase on test data created by uniformly sampling the (δ, ρ) plane. The accuracy of the trained networks on validation data is reported in the captions of each figure. For all experiments, the signal dimension $n = 2^{12}$ while m and k are computed according to the specified δ and ρ.

These numerical experiments show that even a simple neural network is able to approximately determine whether or not a given greedy algorithm and (δ, ρ) pairing will result in successful signal recovery. In particular, the yellow regions in the second row of each figure

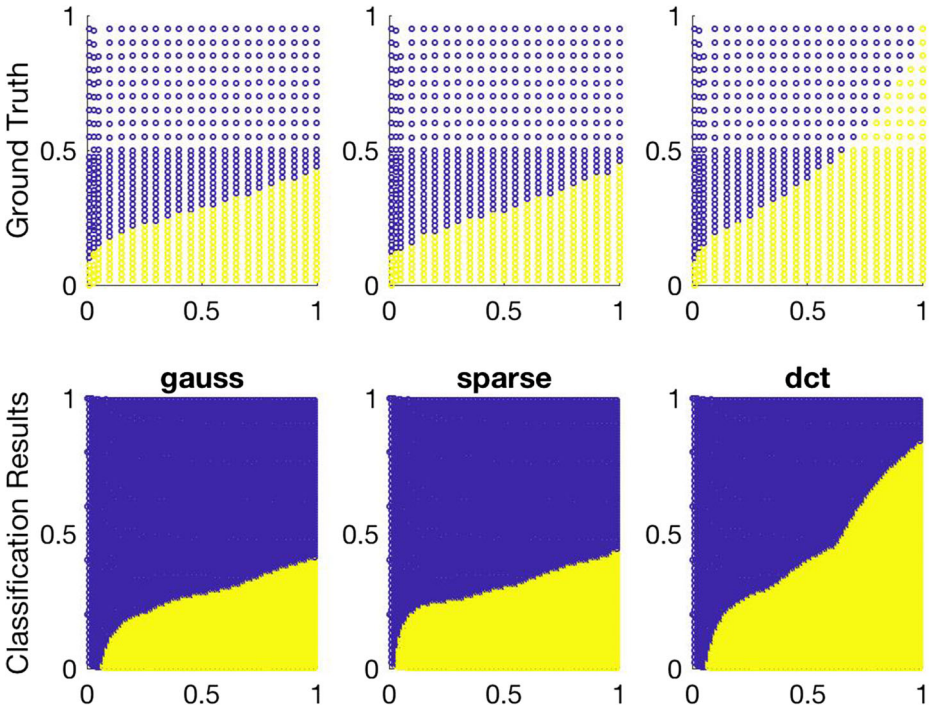

Fig. 2 Recovery for phase transitions of various types of measurement matrices for NIHT. (Test validation accuracy = 0.964)

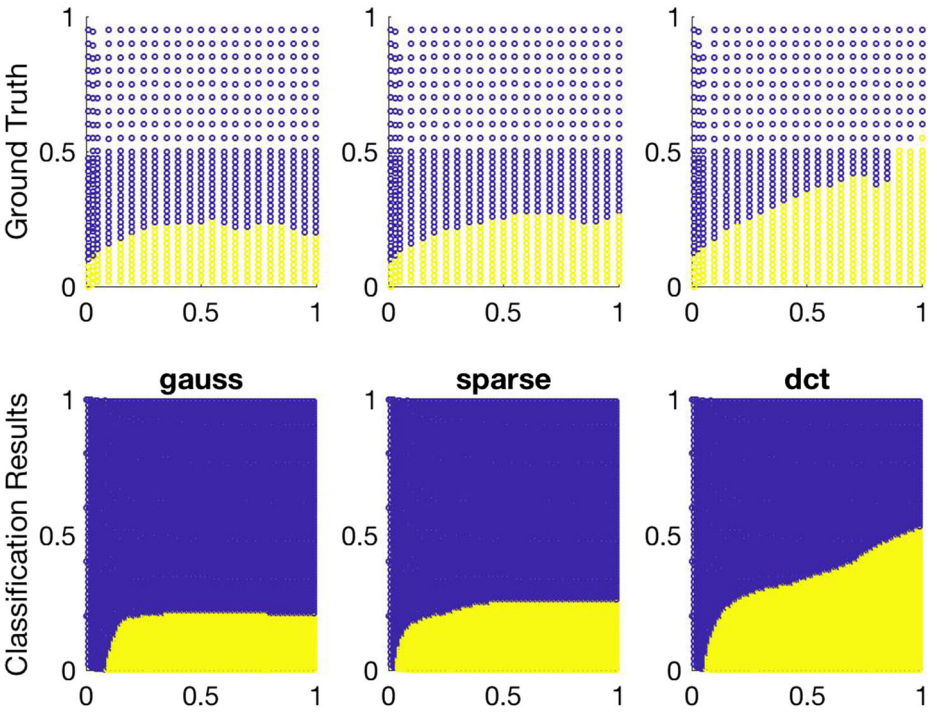

Fig. 3 Recovery for phase transitions of various types of measurement matrices for CSMPSP. (Test validation accuracy = 0.963)

not only roughly approximate the yellow regions in the first row but they also noticeably vary across both algorithm and measurement matrix to match the input training data, as desired.

3 Application II: stochastic gradient descent learning rate selection

We now further test our machine learning framework with an exploration of learning rate schedule selection for the *stochastic gradient descent* (SGD) algorithm. In this set of experiments, we demonstrate that one can use neural networks to select a learning rate schedule which improves the behavior of SGD on a given instance, provided proper training data. After a brief introduction to the vast body of literature regarding the convergence behavior of SGD and corresponding learning rates, we discuss our experimental results and comment on our findings. In Section 3.1, we describe in detail the design of our neural network framework. We additionally describe the construction of the training and testing data provided to the network in each experiment.

SGD is a ubiquitous first-order iterative method for convex optimization. The classical SGD algorithm for optimizing $f(x)$ works as follows: After selecting a learning rate (or step size) schedule α_t and an initialization $\boldsymbol{x} = \boldsymbol{x}_0$, we randomly select an index $i \in \{1, ..., m\}$. While the stopping criteria is not satisfied, we update $\boldsymbol{x}_t = \boldsymbol{x}_{t-1} - \alpha_t \nabla f_i(\boldsymbol{x}_{t-1})$. The applications in which SGD is *la méthode du jour* are diverse and cut across many scientific fields, with perhaps the hottest application currently being in the training of neural networks. The

performance of SGD depends heavily on the selected learning rate (or step size) schedule, $\{\alpha_t\}_{t=1}^{M}$, and parameters of the objective function such as the Lipchitz constant or strong convexity parameter [45, 47, 49, 52]. Parameter tuning SGD can also be interpreted as an algorithm selection problem. There are numerous proposed line search methods for selecting learning rates and methods for performing one-dimensional optimization on the learning rate to speed convergence [18, 42, 44, 55]. In practice, learning rate selection can be quite ad hoc and there are popular heuristics for updating the learning rate [27].

Recently, practitioners and theorists alike have turned their attention to adaptive learning rate schedules, in which the learning rate assigned to a component updates according to information gleaned from the sample [19, 20, 35, 51, 61]. Recent adaptive learning rate approaches approximate Lipschitz parameters and use this to approximately compute a learning rate [47, 57].

Our work presents a machine learning framework which allows practitioners to choose a learning rate schedule without knowledge of objective function parameters. As a proof of concept, we focus on solving least-squares problems, but we stress that our framework could be applied to more complex objective functions. This framework offers practitioners an alternative to heuristics and unknown objective function parameters.

3.1 Experiments

In each of the experiments presented below, we apply SGD to solve a least-squares problem $\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2$ defined by measurement matrix $\boldsymbol{A}$ and measurement vector $\boldsymbol{y}$. The goal of our machine learning framework is to train a neural network to select the optimal learning rate schedule (out of a fixed set of schedules) for a given input linear system represented by its measurement vector, $\boldsymbol{y}$; we specify the measure with which we compare learning rates in each section below. These experiments also require the same three phases as in Section 2: creating training data, training the neural network, and testing the behavior of SGD with the neural network predicted learning rates.

In the first phase, we generate data points consisting of measurement vectors $\boldsymbol{y}$ and labels that indicate the optimal learning rate schedule. We compare only two types of learning rate schedules: the constant learning rate $\alpha_t = c$ and the epoch-based learning rate schedule $\alpha_t = c_1 * c_2^{\lfloor t+1/c_3 \rfloor}$; these constants are defined in each experiment below. To select the optimal learning rate schedule and assign a label to each data point, we run $M = 5000$ iterations of SGD and assign the label of the learning rate schedule that resulted in the smallest recovery error, $\|\boldsymbol{x}_M - \boldsymbol{x}\|_2$ where $\boldsymbol{x}$ is the signal. This phase is completed in MATLAB using version R2017a on a laptop running macOS. In each experiment, our input data points form two classes which correspond to each of the learning rate schedules. The consistent systems are optimally solved with the constant learning rate schedule, while the inconsistent systems are optimally solved with the epoch-based learning rate schedule; this decreasing learning rate schedule helps SGD avoid the larger convergence horizon of the inconsistent systems. These data points are labeled accordingly and the neural network task of predicting the optimal learning rate schedule is equivalent to predicting to which set of systems each data point belongs. In each experiment, the data set consists of 3000 measurement vectors, a portion of which is used for training and the remaining data set is reserved for testing.

In the second phase, we train a neural network with the training portion of the data set, consisting of the measurement vectors $\boldsymbol{y}$ and the optimal learning rate schedule labels for each system. The second phase is performed in Python 3 and Keras 2.1 with TensorFlow as a backend. The neural network architecture we adopt has one hidden layer with 30 nodes. The intuition for this choice of network architecture is that in our experimental setup the

network only needs to determine which systems are consistent; as a linear problem, we expect that a thin, simple architecture should be successful. The hidden layer nodes use ReLU as the activation function and the final output layer uses the sigmoid function. In the experiments below, we sample 75%, 50%, and 25% of the data to train the neural network and reserve the remaining data for testing validation accuracy.

In the third phase, we measure the validation accuracy of the trained neural network predictions on the test set. Additionally, we use the neural network predicted learning rate schedules to solve each least-squares problem in the test set with $M = 5000$ iterations of SGD and measure the resulting average recovery error, $\|\boldsymbol{x}_M - \boldsymbol{x}\|_2$, and average residual error, $\|\boldsymbol{A}\boldsymbol{x}_M - \boldsymbol{y}\|_2$ over the test set. We compare these average error measures for the neural network predicted learning rates with the average errors solving the test set using only the constant learning rate schedule and only the epoch-based learning rate schedule.

3.1.1 Synthetic linear systems

In this experiment, we train a neural network to recommend either the constant learning rate $\alpha_t = 0.01$ or the epoch-based learning rate schedule $\alpha_t = 0.01 * 0.3^{\lfloor t+1/100 \rfloor}$. Here we set $\boldsymbol{A}$ to be a fixed 1000×100 matrix with Gaussian random variable entries drawn *i.i.d.* from $\mathcal{N}(0, 1)$, and we design two types of linear systems with this matrix, consistent and inconsistent. For the set of consistent linear systems, we set $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}/\|\boldsymbol{A}\boldsymbol{x}\|_2$ where $\boldsymbol{x}$ is a Gaussian vector. For the set of inconsistent systems, we set the error $\boldsymbol{e} = \boldsymbol{v} - \boldsymbol{A}(\boldsymbol{A}^*\boldsymbol{A})^\dagger \boldsymbol{A}^*\boldsymbol{v}$ where $\boldsymbol{v}$ is a Gaussian random variable, so that $\boldsymbol{e}$ is orthogonal to the column space of $\boldsymbol{A}$. We then set $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}/\|\boldsymbol{A}\boldsymbol{x}\|_2 + \boldsymbol{e}/\|\boldsymbol{e}\|_2$ and normalize so that $\boldsymbol{y}$ has $\|\boldsymbol{y}\|_2 = 1$. The set of consistent systems are optimally solved with the constant learning rate schedule and the set of inconsistent systems are optimally solved with the epoch-based learning rate schedule.

We train the neural network with random subsets of a collection of 3000 linear system measurement vectors $\boldsymbol{y}$, 1500 of which are consistent and 1500 of which are inconsistent. In our experiment, we measure the average validation accuracy of the neural network predictions on the remaining test measurement vectors for ten trials in which we randomly sample subsets of 75%, 50%, and 25% training data of the 3000 measurement vectors; the average validation accuracies are listed in Table 1. Furthermore, we list the average recovery error,

Table 1 Average test set validation accuracies of trained neural network (averaged over 10 trials), average resulting recovery error $\|\boldsymbol{x}_M - \boldsymbol{x}\|_2$ and residual error $\|\boldsymbol{A}\boldsymbol{x}_M - \boldsymbol{y}\|_2$ on test set for constant learning rate, epoch-based learning rate, and the neural network (NN) predicted learning rates on synthetic linear systems. Smallest error and residual values are bolded for each experiment

Train	Validation	$\|\boldsymbol{x}_M - \boldsymbol{x}\|_2$		
%	Accuracy	Const.	Epoch	NN Pred.
75%	86.00%	0.01142	0.01525	**0.00909**
50%	77.01%	0.01138	0.01530	**0.01064**
25%	66.32%	**0.01116**	0.01524	0.01177

Train	$\|\boldsymbol{A}\boldsymbol{x}_M - \boldsymbol{y}\|_2$		
%	Const.	Epoch	NN Pred.
75%	0.50980	0.64512	**0.45912**
50%	0.50806	0.64590	**0.49707**
25%	**0.49739**	0.64027	0.53053

$\|\boldsymbol{x}_M - \boldsymbol{x}\|_2$, and average residual error, $\|\boldsymbol{A}\boldsymbol{x}_M - \boldsymbol{y}\|_2$, for the approximation computed by $M = 5000$ SGD iterations using first the constant learning rate schedule, then the epoch-based learning rate schedule, and finally the neural network predicted learning rate for each system. These measures are listed in Table 1; the smallest average error is bold-faced in each row. Note that the average recovery error and average residual error for the neural network predicted learning rates are lower than those of the constant learning rate or epoch-based learning rate for the neural networks trained with 75% and 50% of the data. We suspect that the errors associated with the learning rates predicted by the neural network trained with 25% of the data are not the lowest because of the low neural network validation accuracy, which is in turn due to the small amount of training data.

3.1.2 Computerized tomography systems

In this experiment, we again train a neural network to recommend either the constant learning rate $\alpha_t = 0.01$ or the epoch-based learning rate schedule $\alpha_t = 0.01 * 0.95^{\lfloor t+1/100 \rfloor}$. We input two types of linear systems, consistent and inconsistent. Each data point input is the measurement vector $\boldsymbol{y}$ from a computerized tomography system of equations, $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{y}$ (generated by code adapted from the regularization toolbox by PC Hansen [29]). We fix the matrix $\boldsymbol{A}$ to be a CT matrix generated by the command `tomo(20,10)`; here $N = 20$ is the discretization parameter (number of pixels along one edge of the square image) and $f = 10$ is the oversampling factor. This matrix represents the ray directions which are sampled through the signal (image). We then produce consistent CT systems by applying the CT matrix $\boldsymbol{A}$ to the signal $\boldsymbol{x}$, which is an image from the MNIST database [39], producing the measurement vector $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ and then normalizing so that $\|\boldsymbol{y}\|_2 = 1$. These measurement vectors contain a linear combination of the pixels through which the tomography rays pass. This set of systems is optimally solved with the constant learning rate. We produce inconsistent CT systems with error $\boldsymbol{e} = \boldsymbol{v} - \boldsymbol{A}(\boldsymbol{A}^*\boldsymbol{A})^\dagger \boldsymbol{A}^*\boldsymbol{v}$ where $\boldsymbol{v}$ is a Gaussian random variable, so that $\boldsymbol{e}$ is orthogonal to the column space of $\boldsymbol{A}$. The measurement vector $\boldsymbol{y}$ for these inconsistent CT systems is $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}/\|\boldsymbol{A}\boldsymbol{x}\|_2 + 0.5 * \boldsymbol{e}/\|\boldsymbol{e}\|_2$ normalized so that $\|\boldsymbol{y}\|_2 = 1$, where $\boldsymbol{x}$ is an image from the MNIST database. This set of systems is optimally solved with the epoch-based learning rate schedule.

To evaluate these methods, we measure the average validation accuracy of the neural network predictions on the remaining test measurement vectors for ten trials in which we randomly sample subsets of 25%, 50%, and 75% training data of the 3000 measurement vectors; the average validation accuracies are listed in Table 2. Furthermore, we list the average recovery error, $\|\boldsymbol{x}_M - \boldsymbol{x}\|_2$, and average residual error, $\|\boldsymbol{A}\boldsymbol{x}_M - \boldsymbol{y}\|_2$, for the approximation computed by $M = 5000$ SGD iterations using first the constant learning rate schedule, then the epoch-based learning rate schedule, and finally the neural network predicted learning rate for each system. These measures are listed in Table 2; the smallest average error is bold-faced in each row. Note that the average recovery error and average residual error for the neural network predicted learning rates are lower than those of the constant learning rate or epoch-based learning rate, except for the average residual error of the neural network trained with 50% of the data.

In order to visualize the potential improvement offered by using the trained neural net to select optimal step sizes for each tomography system, we plot in Fig. 4 a recovered image using 5000 SGD iterations with each learning rate schedule, and the original image. The neural network predicts the correct optimal learning rate schedule on these systems.

These numerical experiments show that a simple neural network trained with proper training data can predict learning rates which improve the recovery error of SGD on a set of

Table 2 Average test set validation accuracies of trained neural network (averaged over 10 trials), average resulting recovery error $\|\boldsymbol{x}_M - \boldsymbol{x}\|_2$ and residual error $\|\boldsymbol{A}\boldsymbol{x}_M - \boldsymbol{y}\|_2$ on test set for constant learning rate, epoch-based learning rate, and the neural network (NN) predicted learning rates on computerized tomography linear systems. Smallest error and residual values are bolded for each experiment

Train %	Validation Accuracy	$\|\boldsymbol{x}_M - \boldsymbol{x}\|_2$ Const.	Epoch	NN Pred.
75%	88.19%	0.00669	0.00687	**0.00584**
50%	79.68%	0.00669	0.00685	**0.00550**
25%	85.42%	0.00664	0.00683	**0.00538**

Train %	$\|\boldsymbol{A}\boldsymbol{x}_M - \boldsymbol{y}\|_2$ Const.	Epoch	NN Pred.
75%	**0.25087**	0.26671	0.25121
50%	0.25247	0.26792	**0.24717**
25%	0.24885	0.26469	**0.24323**

given systems. We emphasize that this approach is promising for data sets in which knowledge of the data (e.g., consistency of linear systems, approximate Lipschitz parameters, etc.) is limited. Depending upon the makeup of the given data set and the choice of learning rate schedules, choosing to use a single schedule on all data sets may be the optimal choice (in

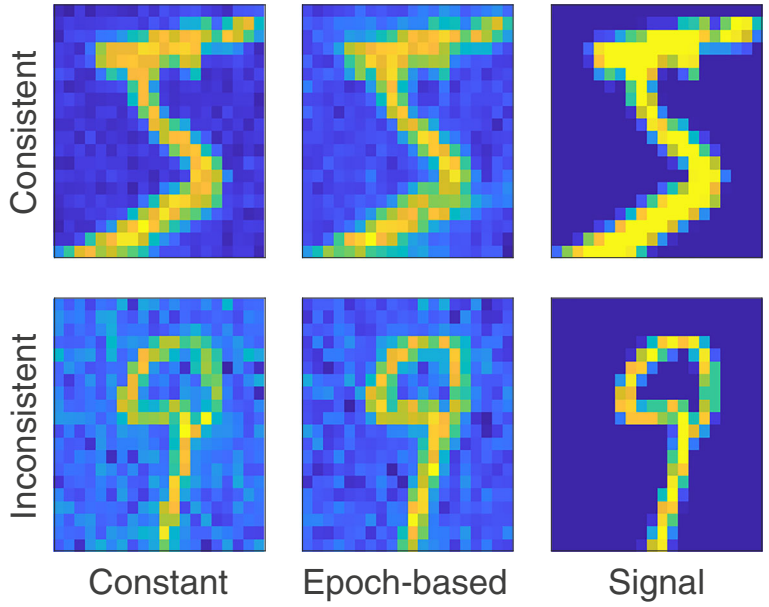

Fig. 4 Recovered signals on consistent system using 5000 SGD iterations with constant learning rate (top left; recovery error 0.00278), epoch-based learning rate (top middle; recovery error 0.00555), and original signal (top right). Recovered signals on inconsistent system using 5000 SGD iterations with constant learning rate (bottom left; recovery error 0.01117), epoch-based learning rate (top middle; recovery error 0.00899), and original signal (bottom right). The neural network correctly predicts the optimal learning rates on these systems

average recovery error), but this approach is useful if you do not have much knowledge about the data set. We illustrate this with a toy situation plotted in Fig. 5. We plot the average recovery error versus the proportion of the test set systems that are inconsistent. For this visualization, we use the recovery errors from the experiment in Fig. 4 to approximate the average recovery errors for each learning rate schedule on each set of systems. For the neural network predicted average recovery error, we assume that the neural network predictions are 80% accurate on both the inconsistent systems and the consistent systems. In this toy example, we see that the neural network predictions outperform the other learning rate schedules when the proportion of inconsistent systems in the test set is between approximately 30% and 80%. However, we additionally note that the neural network predicted learning rates never result in a significantly worse average recovery error than the optimal. Thus, if you know very little about your data set (e.g., how many systems are inconsistent) then our framework offers an efficient method to decrease the resulting average recovery error over all data.

4 Remarks on computational savings

Here, we estimate the computational cost associated to the approach we have proposed and illustrate how it could save computational effort over brute force hyperparameter or algorithm selection. For example, we consider the scenario in which one has N problem instances each of which are represented with m features. Let S denote the number of algorithms or hyperparameters to optimize over to find the most computationally efficient. Now, if the cost of evaluating each algorithm or hyperparameter scales with $f(m, M, \epsilon)$ where M is a cap on the number of iterations and ϵ is a desired solution accuracy, then the cost of the brute force approach to selecting the optimal algorithm or hyperparameter scales with $\mathcal{O}(NSf(m, M, \epsilon))$.

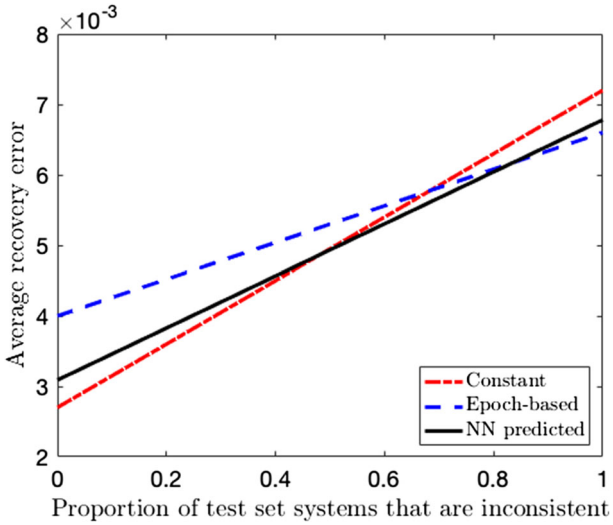

Fig. 5 The average recovery error on the test set versus the proportion of the test set systems that are inconsistent when using only the constant learning rate, only the epoch-based learning rate, or the 80% accurate neural network predicted learning rates

Meanwhile, we can instead analyze our approach with a simple feed-forward neural network with L layers, each with no more than K nodes. Consider the cost of employing the brute-force approach on a fraction β of the N problem instances to form training data, training the neural network with this data, and employing the network to predict on the remaining $(1-\beta)$ fraction of the N problem instances. Forming the training data requires $\mathcal{O}(\beta N S f(m, M, \epsilon))$ effort, the training epochs (backpropagation) requires $\mathcal{O}(\beta N K^2 L^2)$ effort, and predicting (forward propagation) on the remaining data requires $\mathcal{O}((1-\beta)NKL)$; this estimation uses standard complexity bounds for backpropagation and forward propagation (see e.g., [10]). Thus, the effort required by these steps is $\mathcal{O}(\beta N S f(m, M, \epsilon) + N K^2 L^2)$. This can be much more efficient than using the brute-force selection method for all data when the cost of training, $K^2 L^2$, is significantly smaller than the algorithm computational cost, $f(m, M, \epsilon)$.

5 Relevant PAC theorems

Although the focus of our paper has been in training neural networks to select the optimal algorithm or hyperparameter for a specific instance of a given input class, one will notice in our experimental sections that we present the average accuracy of optimal selection over the entire class of inputs in the previous sections. Rather than attempting to learn the optimal algorithm for individual inputs, one instead may appeal to PAC (probably approximately correct) learning for algorithm selection where results guarantee one can learn the optimal algorithm for average behavior over distribution of inputs. Here, instead of selecting the optimal algorithm for a given input, the goal is to select the optimal algorithm for the average performance over a class of inputs. Results in PAC learning provide a selection method to choose an algorithm that, with high probability, is precisely the algorithm that performs best on average in the distribution of instances. For example, in [28], the authors prove that there exists a simple learning method which, with high probability, selects the optimal learning rate for average gradient descent (GD) performance given a distribution over a class of input problems.

5.1 Stochastic gradient descent

In what follows, we prove that there exists a simple learning algorithm which, with high probability, selects nearly the optimal learning rate from an interval of learning rates for the average performance of SGD on an input set of functions which satisfy a standard set of assumptions. In particular, we prove this in the same manner as [28], who prove the parallel result for nonstochastic gradient descent. Our result shows that the trivial learning algorithm $(1+\epsilon, \delta)$-learns the optimal learning rate which minimizes the average number of SGD iterations on the set of input objective functions. Here, the trivial learning algorithm computes the average performance of each learning rate on a sample in a brute force manner and selects the best performing one. This is the learning rate that is then selected, as it is expected to be the best performing on average over the entire distribution. The precise definition of (ϵ, δ)-learning is provided in Definition 1.

Definition 1 [28] The learning algorithm L (ϵ, δ)-learns the optimal algorithm in $\mathcal{A}$ from m samples if, for every distribution $\mathcal{D}$ over Π, with probability at least $1-\delta$ over m samples $x_1, x_2, \cdots, x_m \sim \mathcal{D}$, L outputs an algorithm $\hat{A} \in \mathcal{A}$ with error at most ϵ.

We now introduce our set of input objective functions, assumptions on these functions, the class of algorithms (which are represented simply by their associated learning rate), and additional necessary definitions. Each algorithm in the class of algorithms are defined as SGD with learning rate ρ (denoted $\mathcal{A}_\rho$). Here we consider SGD with fixed learning rate ρ, which is given as above with stopping criterion $\mathbb{E}\|\mathbf{x}_t - \mathbf{x}^*\| \leq \tau$. In detail, the algorithm initializes with $\mathbf{x} = \mathbf{x}_0$, and then randomly selects $i \in \{1, ..., m\}$ and updates $\mathbf{x}_t = \mathbf{x}_{t-1} - \rho \nabla f_i(\mathbf{x}_{t-1})$ until $\mathbb{E}\|\mathbf{x}_t - \mathbf{x}^*\| \leq \tau$. In order to determine the iterate t for which the stopping criteria is satisfied, one can solve for $\mathbf{x}^*$[1], store it, and compute approximation errors at every iteration for different orderings of randomly selected samples. Of course, the number of orderings one must average over depends upon the conditioning of the problem and the concentration of the errors.

Let $\mathcal{D}$ be a distribution over the input set of instances $\mathit{\Pi} = \{(F, \mathbf{x}_0)\}$ where F is the objective function and $\mathbf{x}_0$ is the initial point for SGD. We assume several standard assumptions on the objective function F. In particular, the minimization problem defined by $F(\mathbf{x}) = \sum_{i=1}^m f_i(\mathbf{x})$,

$$\min_{\mathbf{x} \in \mathbb{R}^n} F(\mathbf{x}),$$

has solution $\mathbf{x}^*$ that satisfies $F(\mathbf{x}^*) = \sum_{i=1}^m f_i(\mathbf{x}^*) = 0$ and $\nabla f_i(\mathbf{x}^*) = \mathbf{0}$. Each component function of F is Lipschitz with constant L_i, so

$$\|\nabla f_i(\mathbf{w}) - \nabla f_i(\mathbf{z})\|_2 \leq L_i \|\mathbf{w} - \mathbf{z}\|_2,$$

for all $i = \{1, ..., m\}$ and any $\mathbf{w}, \mathbf{z} \in \mathbb{R}^n$. For example, when solving consistent linear systems, $F(\mathbf{z}) = \frac{1}{2}\sum_{i=1}^m (\langle \mathbf{a}_i, \mathbf{z}\rangle - y_i)^2$ where $\mathbf{a}_i \in \mathbb{R}^n$ then $\nabla f_i(\mathbf{z}) = (\mathbf{a}_i^\top \mathbf{z} - y_i)\mathbf{a}_i$ and one can show

$$L_i = \|\mathbf{a}_i\|_2^2.$$

We define

$$L = \sup_i L_i. \tag{3}$$

In addition, we point out that each component satisfies the co-coercivity property, which guarantees

$$\|\nabla f_i(\mathbf{w}) - \nabla f_i(\mathbf{z})\|_2^2 \leq L_i \langle \mathbf{w} - \mathbf{z}, \nabla f_i(\mathbf{w}) - \nabla f_i(\mathbf{z})\rangle.$$

We additionally assume that there exists a known constant c so that

$$\mathbb{E}_{t-1}\|g^t(\mathbf{x}_0, \rho) - \mathbf{x}^*\| \leq (1 - c)\|\mathbf{x}_{t-1} - \mathbf{x}^*\|, \tag{4}$$

for some known constant c so that $\mathbb{E}\|g^t(\mathbf{x}_0, \rho) - \mathbf{x}^*\| \leq (1 - c)^t X$ where $\|\mathbf{x}_0 - \mathbf{x}^*\| \leq X$ for some initialization $\mathbf{x}_0$. Like [28], we call this the *guaranteed progress* property.

Our goal is now to show that there exists a learning function L that $(1 + \epsilon, \delta)$-learns the algorithm $\mathcal{A}_\rho \in \mathcal{A}$ that minimizes the expected number of iterations required to achieve the

[1] Since the goal of this work is to be able to select the optimal learning rate for average performance over a (possibly extremely large) set of input problems, we consider the cost of this step on the relatively small sample to be acceptable.

stopping criterion on an element of Π sampled according to $\mathcal{D}$. We let $(\mathcal{A}_\eta, \mathbf{x})$ denote the algorithm execution of SGD with fixed step size η and initialization $\mathbf{x}$. We denote the number of iterations required to reach the previously defined stopping criterion $Cost(\mathcal{A}_\eta, \mathbf{x})$, so

$$Cost(\mathcal{A}_\eta, \mathbf{x}) = \min\{t : \mathbb{E}\|\mathbf{x}_t - \mathbf{x}^*\| \leq \tau\}. \tag{5}$$

We let one step of SGD be denoted as

$$g(\mathbf{x}, \rho) := \mathbf{x} - \rho \nabla f_i(\mathbf{x}). \tag{6}$$

Finally, we define

$$D(\rho) = max(\{1, L\rho - 1\}). \tag{7}$$

and

$$H = \log(\tau/LX)/\log(1 - c). \tag{8}$$

We additionally remark that the proof of our main result in this section uses the *pseudo-dimension* of a family of algorithms. The pseudo-dimension naturally extends the VC dimension [56] from binary valued functions to functions that take on real values. This definition is complex and would unnecessarily complicate our manuscript since we only consider the pseudo-dimension of finite sets. For this reason, we leave this definition out and direct readers to Section 3.2 of the excellent paper [28]. Here, we need only the fact that the pseudo-dimension of a finite set of algorithms N is $\log_2 |N|$.

The following lemmas lead up to our main theoretical guarantees for the learnability of optimal step size for SGD and was inspired by the results shown for Gradient Descent (GD) in [28]. In particular, we use a similar approach to [28] where Lemmas 1- 3 are adaptations of Lemmas 3.16-3.18 in [28] from GD to SGD.

The proof of the main theorem uses Lemma 1, which captures the norm difference between one iteration SGD with different initial points and Lemma 2, which captures the norm difference between t iterations of SGD with different initial points and different step sizes. The proof also utilizes Lemma 3, which shows that under mild conditions, the difference between (5) for two different algorithms, $\mathcal{A}_\rho$ and $\mathcal{A}_\eta$, is bounded.

Lemma 1 *For all* $\mathbf{w}, \mathbf{z}$, *and* ρ, *we have*

$$\|g(\mathbf{w}, \rho) - g(\mathbf{z}, \rho)\| \leq D(\rho)\|\mathbf{w} - \mathbf{z}\|,$$

where $g(\mathbf{w}, \rho)$ *and* $D(\rho)$ *are as defined in* (6) *and* (7) *respectively.*

Proof Manipulating the norm yields

$$\begin{aligned}
\|g(\mathbf{w}, \rho) - g(\mathbf{z}, \rho)\|^2 &= \|\mathbf{w} - \rho\nabla f_i(\mathbf{w}) - \mathbf{z} + \rho\nabla f_i(\mathbf{z})\|^2 \\
&= \|\mathbf{w} - \mathbf{z}\|^2 + \rho^2\|\nabla f_i(\mathbf{w}) - \nabla f_i(\mathbf{z})\|^2 - 2\rho\langle \mathbf{w} - \mathbf{z}, \nabla f_i(\mathbf{w}) - \nabla f_i(\mathbf{z})\rangle \\
&\leq \|\mathbf{w} - \mathbf{z}\|^2 + \rho^2\|\nabla f_i(\mathbf{w}) - \nabla f_i(\mathbf{z})\|^2 - \frac{2\rho}{L_i}\|\nabla f_i(\mathbf{w}) - \nabla f_i(\mathbf{z})\|^2 \\
&\leq \|\mathbf{w} - \mathbf{z}\|^2 + \rho^2\|\nabla f_i(\mathbf{w}) - \nabla f_i(\mathbf{z})\|^2 - \frac{2\rho}{L}\|\nabla f_i(\mathbf{w}) - \nabla f_i(\mathbf{z})\|^2 \\
&= \|\mathbf{w} - \mathbf{z}\|^2 + \left(\rho\left(\rho - \frac{2}{L}\right)\right)\|\nabla f_i(\mathbf{w}) - \nabla f_i(\mathbf{z})\|^2
\end{aligned}$$

where the first inequality follows from the co-coercivity property. Now, if $\rho \leq \frac{2}{L}$, then $\|g(\mathbf{w}, \rho) - g(\mathbf{z}, \rho)\| \leq \|\mathbf{w} - \mathbf{z}\|$ so for this case we may set $D(\rho) = 1$. If $\rho > \frac{2}{L}$, we have that

$$\begin{aligned}
\|g(\mathbf{w}, \rho) - g(\mathbf{z}, \rho)\|^2 &\leq \|\mathbf{w} - \mathbf{z}\|^2 + \left(\rho\left(\rho - \frac{2}{L}\right)\right)\|\nabla f_i(\mathbf{w}) - \nabla f_i(\mathbf{z})\|^2 \\
&\leq \|\mathbf{w} - \mathbf{z}\|^2 + \left(\rho\left(\rho - \frac{2}{L}\right)\right)L_i^2\|\mathbf{w} - \mathbf{z}\|^2 \\
&\leq \|\mathbf{w} - \mathbf{z}\|^2 + \left(\rho\left(\rho - \frac{2}{L}\right)\right)L^2\|\mathbf{w} - \mathbf{z}\|^2 \\
&\leq (L\rho - 1)^2\|\mathbf{w} - \mathbf{z}\|^2,
\end{aligned}$$

therefore in this case we may take $D(\rho) = L\rho - 1$, as desired. □

Using the bound on the divergence between SGD steps with the same learning rates provided by Lemma 1, we now bound the divergence between $t+1$ SGD steps with different learning rates beginning at the same point. We show that this is linear in the difference between the learning rates, the constants L, X, and c, and grows with $D(\rho)$.

Lemma 2 *For any t, and $\rho < \eta$, we have*

$$\mathbb{E}\|g^{t+1}(\mathbf{x}, \rho) - g^{t+1}(\mathbf{x}, \eta)\| \leq (\eta - \rho)\frac{D(\rho)^t LX}{c},$$

where $\|\mathbf{x}_0 - \mathbf{x}^*\| \leq X$, $g^{t+1}(\mathbf{x}, \rho) = g(g^{t-1}(\mathbf{x}, \rho), \rho)$ and L, c, $g(\mathbf{x}, \rho)$, and $D(\rho)$ are as defined in (3)*,* (4)*,* (6)*, and* (7) *respectively.*

Proof We first bound $\|g(\mathbf{w}, \rho) - g(\mathbf{z}, \eta)\|$:

$$\begin{aligned}
\|g(\mathbf{w}, \rho) - g(\mathbf{z}, \eta)\| &= \|g(\mathbf{w}, \rho) - \mathbf{z} + \eta\nabla f_i(\mathbf{z}) - \rho\nabla f_i(\mathbf{z}) + \rho\nabla f_i(\mathbf{z})\| \\
&= \|g(\mathbf{w}, \rho) - g(\mathbf{z}, \rho) + (\eta - \rho)\nabla f_i(\mathbf{z})\| \\
&\leq \|g(\mathbf{w}, \rho) - g(\mathbf{z}, \rho)\| + \|(\eta - \rho)\nabla f_i(\mathbf{z})\| \\
&\leq D(\rho)\|\mathbf{w} - \mathbf{z}\| + (\eta - \rho)\|\nabla f_i(\mathbf{z})\|. \\
&= D(\rho)\|\mathbf{w} - \mathbf{z}\| + (\eta - \rho)\|\nabla f_i(\mathbf{z}) - \nabla f_i(\mathbf{x}^*)\|.
\end{aligned}$$

Substituting $\mathbf{w} = g^t(\mathbf{x}, \rho)$ and $\mathbf{z} = g^t(\mathbf{x}, \eta)$:

$$\begin{aligned}
\|g^{t+1}(\mathbf{x}, \rho) - g^{t+1}(\mathbf{x}, \eta)\| &\leq D(\rho)\|g^t(\mathbf{x}, \rho) - g^t(\mathbf{x}, \eta)\| + (\eta - \rho)\|\nabla f_i(g^t(\mathbf{x}, \eta)) - \nabla f_i(\mathbf{x}^*)\| \\
&\leq D(\rho)\|g^t(\mathbf{x}, \rho) - g^t(\mathbf{x}, \eta)\| + (\eta - \rho)L_i\|g^t(\mathbf{x}, \eta) - \mathbf{x}^*\|.
\end{aligned}$$

Using the guaranteed progress assumption and the expectation conditional on the first t iterations, we have:

$$\begin{aligned}
\mathbb{E}\mathbb{E}_t\|g^{t+1}(\mathbf{x}, \rho) &- g^{t+1}(\mathbf{x}, \eta)\| \\
&\leq D(\rho)\mathbb{E}\mathbb{E}_t\|g^t(\mathbf{x}, \rho) - g^t(\mathbf{x}, \eta)\| + (\eta - \rho)L_i\mathbb{E}\mathbb{E}_t\|g^t(\mathbf{x}, \eta) - \mathbf{x}^*\|.
\end{aligned}$$

Iterating the expectation, we have:

$$\begin{aligned}
\mathbb{E}\mathbb{E}_t\|g^{t+1}(\mathbf{x},\rho)-g^{t+1}(\mathbf{x},\eta)\| &\leq D(\rho)\mathbb{E}\|g^t(\mathbf{x},\rho)-g^t(\mathbf{x},\eta)\|+(\eta-\rho)LX(1-c)^t\\
&\leq D(\rho)^t\|\mathbf{x}_0-\mathbf{x}_0\|+(\eta-\rho)LX\sum_{i=0}^{t}D(\rho)^i(1-c)^t\\
&= D(\rho)^t(\eta-\rho)LX\sum_{i=0}^{t}D(\rho)^{-i}(1-c)^i\\
&\leq D(\rho)^t(\eta-\rho)LX\sum_{i=0}^{t}(1-c)^i\\
&= (\eta-\rho)\frac{D(\rho)^tLX}{c}.
\end{aligned}$$

□

Our final lemma uses the fact that the divergence between $t+1$ SGD steps with different learning rates is bounded to show that the cost of the SGD algorithms with these learning rates is "Lipschitz-like." In particular, if the difference between the learning rates is sufficiently small then the difference between the number of iterations necessary for the SGD algorithms is at most one.

Lemma 3 *For all* $\mathbf{x}$ *with* $\|\mathbf{x}-\mathbf{x}^*\|\leq X$, *and* ρ,η *such that* $0\leq\eta-\rho\leq\frac{\tau c^2}{LX}D(\rho)^{-H}$ *we have*

$$|cost(\mathcal{A}_\rho,\mathbf{x})-cost(\mathcal{A}_\eta,\mathbf{x})|\leq 1,$$

where $cost(\mathcal{A},\mathbf{z})$ *is as defined in* (5) *and* τ *is the error tolerance. The variables* L, $D(\rho)$, *and* H *are as defined in* (3), (7), *and* (8) *respectively.*

Proof Assume without loss of generality that $\text{cost}(\mathcal{A}_\eta,\mathbf{x})\leq\text{cost}(\mathcal{A}_\rho,\mathbf{x})$. Define $j=\text{cost}(\mathcal{A}_\eta,\mathbf{x})$ and note that $j\leq H$. By Lemma 3.17 for SGD,

$$\mathbb{E}\|g^j(\mathbf{x},\rho)-g^j(\mathbf{x},\eta)\|\leq\frac{\tau c^2}{LX}D(\rho)^{-H}\frac{D(\rho)^{j-1}LX}{c}\leq\tau c.$$

By the triangle inequality,

$$\mathbb{E}\|g^j(\mathbf{x},\rho)-\mathbf{x}^*\|\leq\mathbb{E}\|g^j(\mathbf{x},\rho)-g^j(\mathbf{x},\eta)\|+\mathbb{E}\|g^j(\mathbf{x},\eta)-\mathbf{x}^*\|\leq\tau c+\tau$$

where the τ term follows from the fact that $j=\text{cost}(\mathcal{A}_\nu,\mathbf{x})$.

Now, note that if we have $\mathbb{E}\|g^j(\mathbf{x},\rho)-\mathbf{x}^*\|\leq\tau$ then $\text{cost}(\mathcal{A}_\rho,\mathbf{x})=j=\text{cost}(\mathcal{A}_\eta,\mathbf{x})$. Thus, we assume that we have not reached the stopping criterion and therefore $\mathbb{E}\|g^j(\mathbf{x},\rho)-\mathbf{x}^*\|>\tau$. By the guaranteed progress in expectation condition, we have $\mathbb{E}\|g^{j+1}(\mathbf{x},\rho)-\mathbf{x}^*\|\leq(1-c)\mathbb{E}\|g^j(\mathbf{x},\rho)-\mathbf{x}^*\|$, which provides the expected improvement

$$\mathbb{E}\|g^j(\mathbf{x},\rho)-\mathbf{x}^*\|-\mathbb{E}\|g^{j+1}(\mathbf{x},\rho)-\mathbf{x}^*\|\geq c\mathbb{E}\|g^j(\mathbf{x},\rho)-\mathbf{x}^*\|>c\tau.$$

Thus, we have $\text{cost}(\mathcal{A}_\rho,\mathbf{x})=j+1$ since, rearranging, we have

$$\mathbb{E}\|g^{j+1}(\mathbf{x},\rho)-\mathbf{x}^*\|<\mathbb{E}\|g^j(\mathbf{x},\rho)-\mathbf{x}^*\|-c\tau\leq\tau c+\tau-\tau c=\tau.$$

□

Finally, the previous sequence of lemmas motivates us to discretize the learning rate interval, which provides us a finite set of SGD algorithms to consider. Previous results of

[28] show that the optimal algorithm in this finite set of algorithms can be (ϵ, δ)-learned by the trivial learning algorithm with a finite set of samples. We then need only apply our last lemma to show that the cost between any nearby learning rate and one of the learning rates in the discretization is at most one to give that the optimal algorithm in $\mathcal{A}$ can be $(1+\epsilon, \delta)$-learned by the trivial learning algorithm.

Theorem 1 (SGD: learnability of step size in stochastic gradient descent) *Let $\mathcal{A} = \{\mathcal{A}_\rho : \rho \in [\rho_l, \rho_u]\}$ where ρ_l and ρ_u are lower and upper bounds for choices of step size respectively. The trivial learning algorithm $(1+\epsilon, \delta)$-learns the optimal algorithm in $\mathcal{A}$ using*

$$m = O\left(\left(\frac{H}{\epsilon}\right)^2 \max\left\{1, \ln(1/\delta), H\log_2\left(\frac{2\rho_u LXD(\rho_u)}{\tau c^2}\right)\right\}\right),$$

samples from $\mathcal{D}$. Here, τ is the error tolerance, X is the uniform upper bound on the approximation error over all iterates, i.e., X such that for all $\mathbf{x}$ iterates from SGD, $\|\mathbf{x} - \mathbf{x}^*\| \leq X$ and the variables L, c, $D(\rho)$, and H are as defined in (3), (4), (7), and (8).

Proof Let $N = \{\rho_i := \rho_l + i\frac{\tau c^2}{LX} D(\rho_u)^{-H} : \rho_i \leq \rho_u\} \cup \{\rho_u\}$. Note that since $D(\rho)$ is an increasing function, we have $\frac{\tau c^2}{LX} D(\rho_u)^{-H} \leq \frac{\tau c^2}{LX} D(\rho)^{-H}$ for all $\rho \in [\rho_l, \rho_u]$. Thus, for any $\rho \in [\rho_l, \rho_u]$ there exists $\rho^* \in N$ such that $|\rho^* - \rho| \leq \frac{\tau c^2}{LX} D(\rho)^{-H}$.

Now, since $\mathcal{A}_N$ is a finite set, the pseudo-dimension of $\mathcal{A}_N = \{\mathcal{A}_\rho : \rho \in N\}$ is at most $\log|N|$. Let L_N be the trivial learning algorithm that returns the algorithm from $\mathcal{A}_N$ that has the best average performance on our sampled subset of Π. Then, by Corollary 3.4 of [28], L_N (ϵ, δ)-learns the optimal algorithm in $\mathcal{A}_N$ using

$$m = O\left(\left(\frac{H}{\epsilon}\right)^2 (d_N + \ln(1/\delta))\right),$$

samples.

Finally, Lemma 3 guarantees that for every ρ there exists $\eta \in N$ so that the difference in the expected costs of $\mathcal{A}_\rho$ and $\mathcal{A}_\eta$ is less than or equal to 1. Thus, the learning algorithm L_N $(1+\epsilon, \delta)$-learns the optimal algorithm in all of $\mathcal{A}$ using

$$m = O\left(\left(\frac{H}{\epsilon}\right)^2 (d_N + \ln(1/\delta))\right),$$

samples.

Now, we simply manipulate $\left(\frac{H}{\epsilon}\right)^2 (d_N + \ln(1/\delta))$. Note that

$$\begin{aligned}
\left(\frac{H}{\epsilon}\right)^2 (d_N + \ln(1/\delta)) &\leq \left(\frac{H}{\epsilon}\right)^2 (\log_2|N| + \ln(1/\delta)) \\
&\leq \left(\frac{H}{\epsilon}\right)^2 \left(\log_2\left(\rho_u / \left(\frac{\tau c^2}{LXD(\rho_u)^H}\right) + 1\right) + \ln(1/\delta)\right) \\
&\leq \left(\frac{H}{\epsilon}\right)^2 \left(\max\left\{H\log_2\left(\frac{2\rho_u LXD(\rho_u)}{\tau c^2}\right), 1\right\} + \ln(1/\delta)\right) \\
&\leq 2\left(\frac{H}{\epsilon}\right)^2 \max\left\{H\log_2\left(\frac{2\rho_u LXD(\rho_u)}{\tau c^2}\right), 1, \ln(1/\delta)\right\}.
\end{aligned}$$

Thus, we have our desired bound on the sample size. □

5.2 Compressed sensing

For the CS application, there are a finite number of algorithms to select from therefore, by Corollary 3.4 of [28], we can conclude the following.

Remark 1 Fix parameters $\epsilon > 0$, $\delta \in (0, 1]$, a set of problem instances $\prod$, and a performance measure cost, COST. The set $\mathcal{A}$ is the set of algorithms (HTP, NIHT, and CSMPSP) with pseudo-dimension at most $d = \log_2 |\mathcal{A}|$ and the trivial algorithm $(2\epsilon, \delta)-$learns the optimal algorithm from $\mathcal{A}$ from $m \geq c \left(\frac{H}{\epsilon}\right)^2 \left(d + \ln\left(\frac{1}{\delta}\right)\right)$ samples where c is a constant and H is the upper bound on COST.

6 Conclusion

We have presented a simple machine learning data-driven approach for empirical algorithm selection or parameter tuning that is widely applicable. Given data and a collection of algorithms or parameters from which to choose, our empirical algorithm selection and tuning approach can be utilized to obtain automatic recommendations. We showcased its broad potential by applying it to compressed sensing and stochastic gradient descent. Additionally, we proved PAC theorems for these problems of interest. More specifically, we showed that there exists a learning algorithm which, with high probability, selects the algorithm that optimizes the average performance on an input set of problem instances with a given distribution.

We must of course address the disadvantages of empirical algorithm selection. Our approach does not formally prove our selection is optimal for all input instances, but instead only with respect to available data. In addition, due to limitations in the existing theory of machine learning, there is no theoretical recipe to choose the features or NN architecture used during training. However, we find it encouraging that these limitations are not so much limitations of our approach, but are standard drawbacks of the theory of machine learning. All of these disadvantages present interesting mathematical directions that are actively being pursued.

On the other hand, there are multiple advantages to using our approach. Foremost, it is very basic and simple but yields advantageous results. A user does not require expert level knowledge of optimization algorithms to make good decisions. Our approach is a pragmatic way to justify algorithmic choices based on available data, and we provide some level of consistency and rigor for evaluating algorithms' performance. Moreover, human experts tend to narrow algorithmic choices to one popular setup which leads to a one-size-fits-all situation. Our approach allows variability of the choice of algorithm or parameters depending on the concrete instance and, most importantly, results in clear improvement of running times or computational cost (measured in number of iterations).

As long as we have a choice of algorithms, or parameter values that determine the behavior of an algorithm, the same process of training a neural network can be used to obtain automatic recommendations. This basic approach presents the possibility that in the future, software will integrate some way of collecting data in order to improve itself. Futuristic code will adapt its own parameters based on historic experience and executions of prior instances. We predict this will be useful in the self-improvement of machine learning techniques and thus in related fields such as computational mathematics that utilize machine learning.

Acknowledgments This material was supported the National Science Foundation grant number DMS-1440140 while the authors were in residence at the Mathematical Science Research Institute in Berkeley, California, during the Fall 2017 semester. De Loera was funded by NSF DMS-1522158, NSF DMS-1818969, and NSF TRIPODS grant (NSF Award no. CCF-1934568). Needell was funded by NSF CAREER DMS-1348721 and NSF BIGDATA 1740325.

References

1. Alvarez, A.M., Louveaux, Q., Wehenkel, L.: A machine learning-based approximation of strong branching. INFORMS J. Comput. **29**(1), 185–195 (2017)
2. Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M.W., Pfau, D., Schaul, T., Shillingford, B., De Freitas, N.: Learning to learn by gradient descent by gradient descent. In: Adv. Neur. In., pp. 3981–3989 (2016)
3. Balcan, M., Dick, T., Sandholm, T., Vitercik, E.: Learning to branch. In: Int. Conf. Mach. Learn., pp. 353–362 (2018)
4. Balcan, M., Nagarajan, V., Vitercik, E., White, C.: Learning-theoretic foundations of algorithm configuration for combinatorial partitioning problems. In: Proc. Conf. Learn. Th., pp. 213–274 (2017)
5. Balte, A., Pise, N., Kulkarni, P.: Meta-learning with landmarking: A survey. Int. J. Comput. Appl. **105**(8) (2014)
6. Bardenet, R., Brendel, M., Kégl, B., Sebag, M.: Collaborative hyperparameter tuning. In: International conference on machine learning, pp. 199–207 (2013)
7. Bengio, Y., Lodi, A., Prouvost, A.: Machine learning for combinatorial optimization: a methodological tour d'horizon. arXiv:1811.06128 (2018)
8. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. J. Mach. Learn. Res. **13**(1), 281–305 (2012)
9. Bertsimas, D., Stellato, B.: The voice of optimization. Mach. Learn., 1–29 (2020)
10. Bishop, C.M.: Pattern Recognition and Machine Learning. Springer, Berlin (2006)
11. Blanchard, J.D., Tanner, J.: Performance comparisons of greedy algorithms in compressed sensing. Numer. Linear Algebr. **22**(2), 254–282 (2015)
12. Blumensath, T., Davies, M.E.: Normalized iterative hard thresholding: Guaranteed stability and performance. IEEE J. Sel. Top. Signa. **4**(2), 298–309 (2010)
13. Bonami, P., Lodi, A., Zarpellon, G.: Learning a classification of mixed-integer quadratic programming problems. In: van Hoeve, W.J. (ed.) Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th International Conference, CPAIOR 2018, Delft (2018). The Netherlands, June 26-29 Proceedings, volume 10848 of Lecture Notes in Computer Science, pp. 595–604. Springer
14. Candes, E.J., Plan, Y.: Matrix completion with noise. Proc. IEEE **98**(6), 925–936 (2010)
15. Candès, E.J., Recht, B.: Exact matrix completion via convex optimization. Found. Comput. Math. **9**(6), 717 (2009)
16. Candès, E.J., Tao, T.: Decoding by linear programming. IEEE T. Inform. Theory **51**, 4203–4215 (2005)
17. Davenport, M., Needell, D., Wakin, M.B.: Signal cosa space MP for sparse recovery with redundant dictionaries. IEEE T. Inform. Theory **59**(10), 6820 (2012)
18. De, S., Yadav, A., Jacobs, D., Goldstein, T.: Big batch SGD: Automated inference using adaptive batch sizes. arXiv:1610.05792 (2017)
19. Défossez, A., Bach, F.: Adabatch: Efficient gradient aggregation rules for sequential and parallel stochastic gradient methods. arXiv:1711.01761 (2017)
20. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res. **12**(7), 2121–2159 (2011)
21. Eggensperger, K., Lindauer, M., Hutter, F.: Neural networks for predicting algorithm runtime distributions. In: Lang, J. (ed.) Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, pp. 1442–1448. Stockholm, Sweden (2018). ijcai.org
22. Eldar, Y.C., Kutyniok, G.: Compressed Sensing: Theory and Applications. Cambridge University Press (2012)
23. Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., Hutter, F.: Efficient and robust automated machine learning. In: Adv. Neur. In., pp. 2962–2970 (2015)
24. Foucart, S.: Hard thresholding pursuit: an algorithm for compressive sensing. SIAM J. Numer. Anal. **49**(6), 2543–2563 (2011)
25. Foucart, S., Rauhut, H.: A mathematical introduction to compressive sensing, vol. 1. Birkhäuser, Basel (2013)

26. Gu, X., Needell, D., Tu, S.: On practical approximate projection schemes in signal space methods. SIAM Undergraduate Research Online **9**, 422–434 (2016)
27. Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., He, K.: Accurate, large minibatch SGD: training imagenet in 1 hour. arXiv:1706.02677 (2017)
28. Gupta, R., Roughgarden, T.: A PAC approach to application-specific algorithm selection. SIAM J. Comput. **46**(3), 992–1017 (2017)
29. Hansen, P.C.: Regularization tools: a MATLAB package for analysis and solution of discrete ill-posed problems. Numer. Algorithm. **6**(1), 1–35 (1994)
30. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proc. CVPR IEEE, pp. 770–778 (2016)
31. He, Y., Yuen, S.Y.: Black box algorithm selection by convolutional neural network. arXiv:2001.01685 (2019)
32. Huang, G., Liu, Z., Van Der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: Proceedings if CVPR IEEE, pp. 4700–4708 (2017)
33. Khalil, E.B., Dai, H., Zhang, Y., Dilkina, B., Song, L.: Learning combinatorial optimization algorithms over graphs. In: Guyon, I.I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, pp. 6348–6358, Long Beach (2017)
34. Khalil, E.B., Dilkina, B., Nemhauser, G.L., Ahmed, S., Shao, Y.: Learning to run heuristics in tree search. In: Proceedings Int Joint Conf. Artif., pp. 659–666 (2017)
35. Kingma, D.P., Adam, J.B.a.: A method for stochastic optimization. arXiv:1412.6980 (2014)
36. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Adv. Neur. In., pp. 1097–1105 (2012)
37. Kruber, M., Lübbecke, M.E., Parmentier, A.: Learning when to use a decomposition. In: Salvagnin, D., Lombardi, M. (eds.) Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua (2017). Proceedings, volume 10335 of Lecture Notes in Computer Science, pp. 202–210. Springer
38. Lagoudakis, M.G., Littman, M.L.: Algorithm selection using reinforcement learning. In: Int. Conf. Mach. Learn., pp. 511–518 (2000)
39. LeCun, Y., Cortes, C., Burges, C.: The MNIST database of handwritten digits. Available at http://yann.lecun.com/exdb/mnist/, Accessed: 21 Dec 2018 (2010)
40. Leyton-Brown, K., Hoos, H.H., Hutter, F., Xu, L.: Understanding the empirical hardness of NP-complete problems. Commun. ACM **57**(5), 98–107 (2014)
41. Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., Talwalkar, A.: Hyperband: A novel bandit-based approach to hyperparameter optimization. J. Mach. Learn. Res. **18**(185), 1–52 (2018)
42. Mahsereci, M., Hennig, P.: Probabilistic line searches for stochastic optimization. In: Adv. Neur. In., pp. 181–189 (2015)
43. Maleki, A., Donoho, D.L.: Optimally tuned iterative reconstruction algorithms for compressed sensing. IEEE J. Sel. Top Signa. **4**(2), 330–341 (2010)
44. Massé, P.-Y., Ollivier, Y.: Speed learning on the fly. arXiv:1511.02540 (2015)
45. Moulines, E., Bach, F.R.: Non-asymptotic analysis of stochastic approximation algorithms for machine learning. In: Adv. Neur. In., pp. 451–459 (2011)
46. Needell, D., Tropp, J.: CosaMP: Iterative signal recovery from incomplete and inaccurate samples. Appl. Comput. Harmon. A. **26**(3), 301–321 (2009)
47. Needell, D., Ward, R., Srebro, N.: Stochastic gradient descent, weighted sampling, and the randomized Kaczmarz algorithm. In: Adv. Neur. In., pp. 1017–1025 (2014)
48. Pfahringer, B., Bensusan, H., Giraud-Carrier, C.G.: Meta-learning by landmarking various learning algorithms. In: ICML, pp. 743–750 (2000)
49. Robbins, H., Monro, S.: A stochastic approximation method. Ann. Math. Stat. **22**, 400–407 (1951)
50. Rudelson, M., Vershynin, R.: On sparse reconstruction from Fourier and Gaussian measurements. Comm. Pure Appl. Math. **61**, 1025–1045 (2008)
51. Schaul, T., Zhang, S., LeCun, Y.: No more pesky learning rates. In: Int. Conf. Mach. Learn., pp. 343–351 (2013)
52. Shamir, O., Zhang, T.: Stochastic gradient descent for non-smooth optimization Convergence results and optimal averaging schemes. In: Int. Conf. Mach. Learn., pp. 71–79 (2013)
53. Smith, K.A.: Neural networks for combinatorial optimization: a review of more than a decade of research. INFORMS J. Comput. **11**(1), 15–34 (1999)
54. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: Proceedings of CVPR IEEE, pp. 1–9 (2015)

55. Tan, C., Ma, S., Dai, Y.-H., Qian, Y.: Barzilai-Borwein step size for stochastic gradient descent. In: Adv. Neur. In., pp. 685–693 (2016)
56. Vapnik, V.N., Chervonenkis, A.Y.: On the uniform convergence of relative frequencies of events to their probabilities. In: Measures of Complexity, pp. 11–30. Springer (2015)
57. Wu, X., Ward, R., Bottou, L.: WNGrad Learn the learning rate in gradient descent. arXiv:1803.02865 (2018)
58. Yang, C., Akimoto, Y., Kim, D.W., Udell, M.: OBOE: Collaborative filtering for AutoML initialization. In: Proceedings of 25th ACM SIGKDD International Conf. Knowledge Discovery & Data Mining, pp. 1173–1183 (2019)
59. Yang, Y., Zhong, Z., Shen, T., Lin, Z.: Convolutional neural networks with alternately updated clique. In: Proceedings of CVPR IEEE, pp. 2413–2422 (2018)
60. Yao, Q., Wang, M., Chen, Y., Dai, W., Yi-Qi, H., Yu-Feng, L., Wei-Wei, T., Qiang, Y., Yang, Y.: Taking human out of learning applications A survey on automated machine learning. arXiv:1810.13306 (2018)
61. Zeiler, M.D.: ADADELTA: an adaptive learning rate method. arXiv:1212.5701 (2012)