Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.Doi Number

Progress Indication for Deep Learning Model Training: A Feasibility Demonstration

Qifei Dong[1] and Gang Luo[1]

[1]Department of Biomedical Informatics and Medical Education, University of Washington, Seattle, WA 98195, USA

Corresponding author: Gang Luo (luogang@uw.edu).

Gang Luo was partially supported by the National Heart, Lung, and Blood Institute of the National Institutes of Health under Award Number R01HL142503. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

ABSTRACT Deep learning is the state-of-the-art learning algorithm for many machine learning tasks. Yet, training a deep learning model on a large data set is often time-consuming, taking several days or even months. During model training, it is desirable to offer a non-trivial progress indicator that can continuously project the remaining model training time and the fraction of model training work completed. This makes the training process more user-friendly. In addition, we can use the information given by the progress indicator to assist in workload management. In this paper, we present the first set of techniques to support non-trivial progress indicators for deep learning model training when early stopping is allowed. We report an implementation of these techniques in TensorFlow and our evaluation results for both convolutional and recurrent neural networks. Our experiments show that our progress indicator can offer useful information even if the run-time system load varies over time. In addition, the progress indicator can self-correct its initial estimation errors, if any, over time.

INDEX TERMS Deep learning, model training, progress indicator, TensorFlow

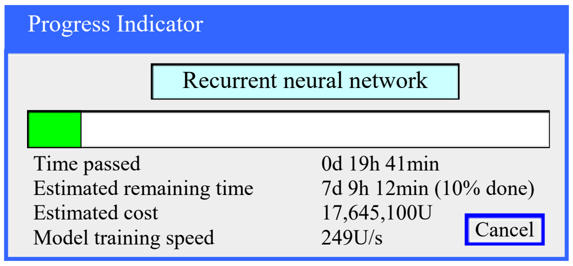
LIST OF SYMBOLS

$\lfloor \rfloor$	Floor function.
a	Scaling factor of the inverse power-law function.
b	Exponent of the inverse power-law function.
b_{max}	Maximum number of batches allowed for model training.
B	Number of training instances in each batch.
c	Bias term of the inverse power-law function.
c_γ	Coefficient used to compute γ.
d	Number of input variables of the objective function.
e_i	Validation error at the i-th validation point.
$f(i)$	Regression function's value at the i-th validation point.
g	Number of batches of model training between two consecutive validation points.
$h(n)$	Sequence number of the current validation point on the current segment.
I_{max}	Maximum number of rounds allowed for the inner loop in each round of the outer loop of the truncated Newton method
k	Number of synthetic validation curves generated via Monte Carlo simulation at each validation point after the τ_v-th one.
K	Size of the sliding time window used for computing the model training speed.
l_i	Number of validation points on the i-th segment of the validation curve.
m_e	Maximum number of epochs allowed for model training.
n	Number of validation points obtained thus far.
n_i	A simulated random noise at the i-th validation point.
p	Patience.
q_{j-1}	Sequence number of the last validation point on the previous segment of the validation curve.
r	Number of disjoint intervals into which the possible range of the simulated number of validation points needed for model training is divided.
R_{max}	Maximum number of rounds allowed for the loop.
$s(i)$	Sequence number of the segment of the validation curve that the i-th validation point is on.
t	Threshold used to identify the initial transient stage of each segment of the validation curve.
T	Number of data instances in the training set.
T_{max}	Maximum number of rounds allowed for the outer loop of the truncated Newton method.

U	Unit of work.
V	Number of data instances in the validation set.
$v_{j\text{-}1}$	Number of validation points needed for model training that is estimated at the last validation point on the previous segment of the validation curve.
v_{max}	Maximum number of validation points allowed for model training.
w	Maximum number of validation points allowed to fit the regression function.
w'	Number of validation points used to fit the regression function.
α	Initial learning rate used in the exponential decay schedule.
α_i	Learning rate at the *i*-th validation point.
β_i	Learning rate on the *i*-th segment of the validation curve.
γ	Threshold used to decide which interval is regarded as a local mode.
δ	min_delta.
ε	Tolerance.
ρ	Constant controlling the learning rate's decay speed in the exponential decay schedule.
$\hat{\sigma}^2$	Estimated variance of the random noise when a fixed learning rate is used during the entire model training process.
$\hat{\sigma}_i^2$	Estimated variance of the random noise at the *i*-th validation point.
$\hat{\sigma}_{(i)}^2$	Estimated variance of the random noise on the *i*-th segment of the validation curve.
τ_v	Threshold on the number of validation points reached, beyond which we use the validation curve to refine the projected number of validation points needed for model training.

I. INTRODUCTION

The need for non-trivial progress indicators for deep learning model training


Progress Indicator

Recurrent neural network

Time passed 0d 19h 41min
Estimated remaining time 7d 9h 12min (10% done)
Estimated cost 17,645,100U
Model training speed 249U/s

FIGURE 1. A progress indicator for deep learning model training.

Deep learning is the state-of-the-art learning algorithm for many machine learning tasks like image classification, natural language processing, and speech recognition [1]. But, building a deep learning model on a large data set is often time-consuming. Using 50 graphics processing units (GPUs), a Google team spent two months training a deep neural network on 300 million images [2]. With 200 central processing units, Weyand *et al*. [3] took 2.5 months to train a convolutional neural network on 126 million photos. Akiba *et al*. [4] showed that 29 hours were needed to train a convolutional neural network on the ImageNet data set [5] with two GPUs. 15 minutes were needed with 1,024 GPUs. As a standard human-computer interaction principle, for each task running longer than 10 seconds, we need a non-trivial progress indicator (see Fig. 1) to continuously project the remaining task running time and the fraction of the task completed [6, Ch. 5.5]. Thus, progress indicators are desirable for deep learning model training.

Besides making the deep learning model training process more user-friendly, we can use the information given by the progress indicator to assist with workload management as outlined in our papers [7], [8]. We recently talked with Yasser M. Ibrahim, the head of distributed machine learning at Amazon. He mentioned that using a large computer cluster, his team took several months to train a deep neural network supporting Alexa's speech recognition function. Every so often, his team retrains this neural network and would like to finish the re-training in a given amount of time. As the amount of training data, the neural network's hyper-parameter values, and the server capacity continue changing over time, his team needs a method to find an appropriate cluster configuration for each round of re-training. A workload management approach aided by progress indicators would serve this purpose [7].

A neural network is trained in one or more epochs, each of which requires going through all of the training instances once. Some deep learning software supplies trivial progress indicators during model training, e.g., by displaying the number of epochs that has been completed [9] or the value of the objective function achieved [10] over time. Yet, this information is too coarse-grained for many purposes. On a large data set, a large amount of time is needed to go through an epoch. Moreover, early stopping is widely used in deep learning model training to help avoid overfitting. When early stopping is allowed, the number of epochs needed for model training is unknown beforehand, but dynamically decided during model training based on some stopping criterion [1]. Our prior work [11] presents a technique to support non-trivial progress indicators for deep learning model training when the number of epochs needed for model training is known beforehand. This technique updates the projected numbers for the model training task once every few seconds, but is unable to handle early stopping. To the best of our knowledge, no other technique has been published to offer non-trivial progress indicators for deep learning model training. How to support such progress indicators in the presence of early stopping remains an open problem.

Our contributions

To address the gap, in this paper we present the first set of techniques to support non-trivial progress indicators for deep learning model training when early stopping is allowed. With low overhead, our techniques can handle various combinations of the deep learning model, the learning rate schedule like learning rate decay, and the optimization method [12].

A deep learning model is trained in batches. In each batch, a fixed number of training instances are used to compute the updates to the model's parameters. Each batch's running cost is relatively stable and can be quickly measured. Thus, the key to estimating the progress of model training is to project the number of batches needed for model training. During model training, we use the non-smooth learning curve on the validation set (a.k.a. the validation curve) to make this projection. This curve depicts the model's error rates on the validation set, i.e., the validation errors, obtained over time. As Fig. 2 shows, the validation error tends to reduce over time before early stopping occurs and also oscillates over time. If we use a monotonically decreasing function to model the validation curve without accommodating the oscillations, and directly apply the early stopping criterion to the projected curve, we seldom obtain a good estimate of the number of batches needed for model training. To address this challenge, we regard the validation curve as the sum of a smooth trend curve and some zero-mean random noise. We use a regression function to estimate the trend curve, and historical data to gauge the random noise's variance. If the learning rate changes over time, we also model the change's impact on the random noise's variance. Then we use a Monte Carlo simulation approach to project the number of batches needed for model training. By adding simulated random noise to the projected trend curve, we generate several synthetic validation curves. On each of them, we apply the early stopping criterion to obtain a simulated number of batches needed for model training. The estimated mode of these simulated numbers forms the basis for the projected number of batches needed for model training. To the best of our knowledge, this is the first time Monte Carlo simulation has been used for progress indication and is a main innovation of this work.

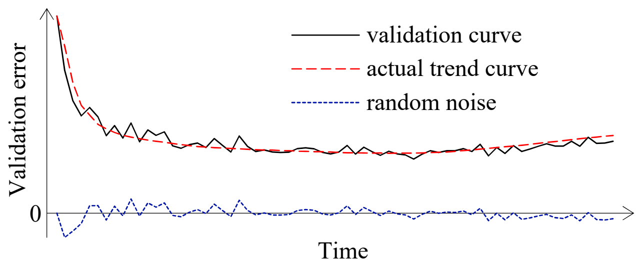


FIGURE 2. The validation curve = a trend curve + some random noise.

We implemented our techniques in TensorFlow [13], an open-source deep learning software package. We report our evaluation results for both convolutional and recurrent neural networks. Our results show that with negligible run-time overhead, the resulting progress indicator can provide useful information even in the presence of varying run-time system loads. Also, the progress indicator can self-correct its initial estimation errors, if any, over time.

Organization of the paper

The rest of the paper is organized as follows. Section II reviews the related work. Section III describes our proposed techniques for implementing progress indicators for deep learning model training when early stopping is allowed. Section IV reports an implementation of our techniques in Tensorflow, as well as the performance evaluation results of the resulting progress indicators. Section V presents some interesting areas for future work. Section VI concludes the paper.

II. RELATED WORK

In this section, we briefly review the related work. A detailed discussion of the related work is available in our prior paper [7].

Sophisticated progress indicators

For machine learning model training, we have built sophisticated progress indicators for decision tree, random forest, and neural network when the number of epochs needed for model training is known beforehand [7], [11]. In addition, sophisticated progress indicators have been proposed for database queries [8], [14]-[17], static program analysis [18], program compilation [19], subgraph queries [20], MapReduce jobs [21], [22], and automatic machine learning model selection [23], [24]. As each kind of task has its own unique properties, we cannot directly adopt the existing techniques [7], [8], [11], [14]-[24] to implement progress indicators for deep learning model training when early stopping is allowed.

Estimating deep learning model training time

Justus *et al.* [25] proposed a meta learning method for estimating an epoch's running time before starting to train a deep learning model, by adopting features of the model, the computational resources, and the training data set used to train another deep learning model. This method predicts neither the number of epochs nor the time needed for model training.

For estimating a deep learning model's training time before model training starts, researchers have proposed several methods including Bayesian optimization [26], meta learning using Multivariate Adaptive Regression Splines [27], meta learning via support vector regression [28], and meta learning via polynomial regression [29]. The projected numbers are frequently inaccurate, are not continuously refined, and could differ significantly from the true model training time on a loaded computer. To build a non-trivial progress indicator, we need to continuously refine the projected model training time.

Complexity analysis for training neural networks

Much research has been done on computing the time complexity of training a neural network [30, Ch. 24], [31], [32]. Yet, this information is not enough for constructing progress indicators and provides no projected model training time on a loaded computer. Time complexity typically ignores data properties affecting the model training cost, as well as the lower order terms and coefficients required for predicting the model training cost. An ideal progress indicator should continuously refine the model training cost as model training proceeds.

III. IMPLEMENTATION TECHNIQUES

In this section, we describe our techniques for implementing progress indicators for deep learning model training when early stopping is allowed. Section III-A introduces some concepts and notations that will be used throughout this paper. Section III-B gives an overview of our progress indication method. Sections III-C to III-E show how to estimate the number of batches needed for model training when a fixed learning rate is used during the entire model training process, when a continuous decay schedule for the learning rate is used, and when a step decay schedule for the learning rate is used, respectively. Section III-F discusses the computational complexity of estimating the number of batches needed for model training.

A. SOME CONCEPTS AND NOTATIONS

In this section, we introduce some concepts and notations that will be used throughout this paper. We have two pre-set positive integers B and g, as well as a given early stopping criterion. A deep learning model is trained in batches. In each batch, B training instances are used to compute the updates to the model's parameters. After every g batches of model training, we reach a validation point. At that time, we compute the model's error rate on the validation set, i.e., the validation error, and check whether the early stopping criterion is met. If so, model training is ended.

The validation curve depicts the validation errors obtained over time during model training. Many early stopping criteria exist, most of which are based on the validation curve [1], [33]-[35]. One criterion is to stop model training when the validation error has not improved over the best one recorded for a given number of validation points [1], [33]. Another criterion adopts the idea of stopping model training when the validation error is over the best one recorded by at least a given threshold, while the model's error rate on the training set no longer improves much [33]. Duvenaud *et al* [34] proposed a criterion based on estimating the log marginal likelihood without using a validation set. Mahsereci *et al*. [35] proposed a criterion based on some local statistics of the computed gradients without using a validation set.

The goal of this paper is neither to handle all of the existing early stopping criteria nor to make the progress indicator's projections reach the maximum possible accuracy. Instead, our goal is to demonstrate via a case study, the feasibility of providing non-trivial and useful progress indication for deep learning model training when early stopping is allowed. Frequently, users can benefit from a rough estimate of the remaining model training time [36]. Our demonstration focuses on a widely used early stopping criterion with two pre-set numbers: min_delta $\delta \geq 0$ and patience $p>0$ [37]. The criterion is met if the validation error improves by $\leq\delta$ for p validation points consecutively. That is, letting e_i denote the validation error at the i-th validation point, model training stops at the s-th validation point if $e_{s-p}-e_j<\delta$ holds for each of $j=s-p+1$, $s-p+2$, …, and s.

B. OVERVIEW OF OUR PROGRESS INDICATION METHOD

In this section, we give an overview of our progress indication method. We start with an initial estimate of the model training cost. Both the predicted model training cost and the current model training speed are gauged by U, the unit of work. Each U depicts the average amount of work needed for processing each training instance once in two steps in model training. The first step is to go forward through the neural network once to compute its prediction result on the training instance. The second step is to go backwards through the neural network once for backpropagation.

During model training, we keep gathering multiple statistics, such as the number of batches done, and use them to keep refining the estimated model training cost. We keep checking the model training speed defined as the number of Us completed per second during the K seconds before the current time point. By default, K's value is 10. At any moment,

the projected remaining model training time
= the projected remaining model training cost / the current model training speed.

Periodically, we update the progress indicator with the latest information. As the model training task keeps running, we gather more precise information of it. As a result, our estimates tend to become increasingly accurate over time.

Computing the model training cost

The model training cost is dominated by two components and can be roughly regarded as their sum. The first component is the cost of processing the training instances. The second component is the cost of computing the validation errors. The first one is easy to compute.

The cost of processing the training instances
= the number of batches needed for model training × the number of training instances per batch × the average amount of work needed for processing a training instance once in model training
= the number of batches needed for model training$\times B\times 1$
= the number of batches needed for model training$\times B$.

Next, we compute the second component. We call each data instance in the validation set a validation instance.

The cost of computing the validation errors
= the number of validation points needed for model training × the number of data instances in the validation set × the average amount of work needed for processing a validation instance once to compute the validation error.

To process a validation instance once, we need to go forward through the neural network once to compute its prediction result on the validation instance. We use the number of multiplication operations needed to estimate the processing cost [31]. Each neuron typically takes multiple inputs, each of which links to a distinct connection weight. When going forward through the neural network, we need to compute the neuron's output by multiplying each input by its linked connection weight. In comparison, when going backwards through the neural network, we need to compute a partial derivative with respect to each input and a partial derivative with respect to each connection weight. The former step requires doing a multiplication with the connection weight linked to the input. The latter requires doing a multiplication with the input linked to the connection weight. Hence, as a rough approximation based on the number of multiplication operations needed, we regard the cost of going backwards through the neural network once to be twice that of going forward through the neural network once. That is, the average amount of work needed for processing a validation instance one time = $U/3$. Consequently,

the cost of computing the validation errors
= the number of validation points needed for model training$\times V/3$,

with V being the number of data instances in the validation set.

Summing the two components, we have

the model training cost
= the number of batches needed for model training$\times B$ + the number of validation points needed for model training$\times V/3$.

Before model training starts, we can easily know B and V's values. Thus, to estimate the model training cost, we mainly need to estimate the number of batches and the number of validation points needed for model training.

Let T denote the number of data instances in the training set. Before a deep neural network is trained, the user of the deep learning software needs to specify the value of a hyper-parameter m_e showing the maximum number of epochs allowed for model training. Each epoch requires passing through all of the training instances once and includes T/B batches of model training. The maximum number of batches allowed for model training is

$$b_{max}=m_e\times T/B.$$

Before model training starts, we can easily know T and B's values and subsequently b_{max}'s value. Recall that a validation point is reached every g batches of model training. If early stopping occurs before finishing the b_{max}-th batch,

the number of batches needed for model training
= the number of validation points needed for model training$\times g$.

If early stopping never occurs and model training reaches the maximum number of batches allowed,

the number of batches needed for model training = b_{max},

and

the number of validation points needed for model training
$= v_{max}$
$\stackrel{\text{def}}{=} \lfloor b_{max}/g \rfloor$.

Here, $\lfloor\rfloor$ is the floor function, e.g., $\lfloor 3.4 \rfloor=3$. v_{max} is the maximum number of validation points allowed for model training. Thus, the key to estimating the model training cost is to estimate the number of validation points needed for model training, and subsequently, whether early stopping will ever occur.

Estimating the number of validation points needed for model training

Initially, with no extra information, we estimate the number of validation points needed for model training to be v_{max}, the maximum number of validation points allowed for model training. During model training, once the number of validation points reached is $\geq$ a given threshold τ_v, we start using the validation curve to keep refining the projected number of validation points needed for model training. In our implementation, we choose 3 as τ_v's default value to strike a balance between having enough validation points to make a reasonable projection and not having to wait too long before the initial projected number could be refined.

As Fig. 2 shows, the validation curve often oscillates over time. We regard it as the sum of a smooth trend curve and some zero-mean random noise. At each validation point that is after the τ_v-th one and where the early stopping criterion is unmet, we first fit a smooth regression function to the validation curve up to this point, and then use the fitted function to estimate the trend curve beyond this point. Since the regression function is smooth, the estimated trend curve does not reflect the oscillations on the validation curve. Thus, directly applying the early stopping criterion to the estimated trend curve often does not lead to a good estimate of the number of validation points needed for model training. For example, as the validation error tends to decrease over time, we use a monotonically decreasing regression function to estimate the trend curve. When the min_delta δ=0, the early stopping criterion includes a term that the validation error increases at some point. Thus, the criterion is never met on the estimated trend curve, even if early stopping occurs frequently in practice.

To address this issue, we use historical data to gauge the random noise's variance. Then we use a Monte Carlo simulation approach to project the number of validation points needed for model training. By adding simulated random noise to the projected trend curve, we generate several synthetic validation curves. To each of them, we apply the early stopping criterion and obtain the number of validation points needed. The smaller of this number and v_{max}, the maximum number of validation points allowed for model training, becomes a simulated number of validation points needed for model training. The estimated mode of these simulated numbers forms the basis for our projected number of validation points needed for model training.

Fig. 3 shows the flow chart of our method for estimating the number of validation points needed for model training. Sections III-C to III-E present the details of this method.

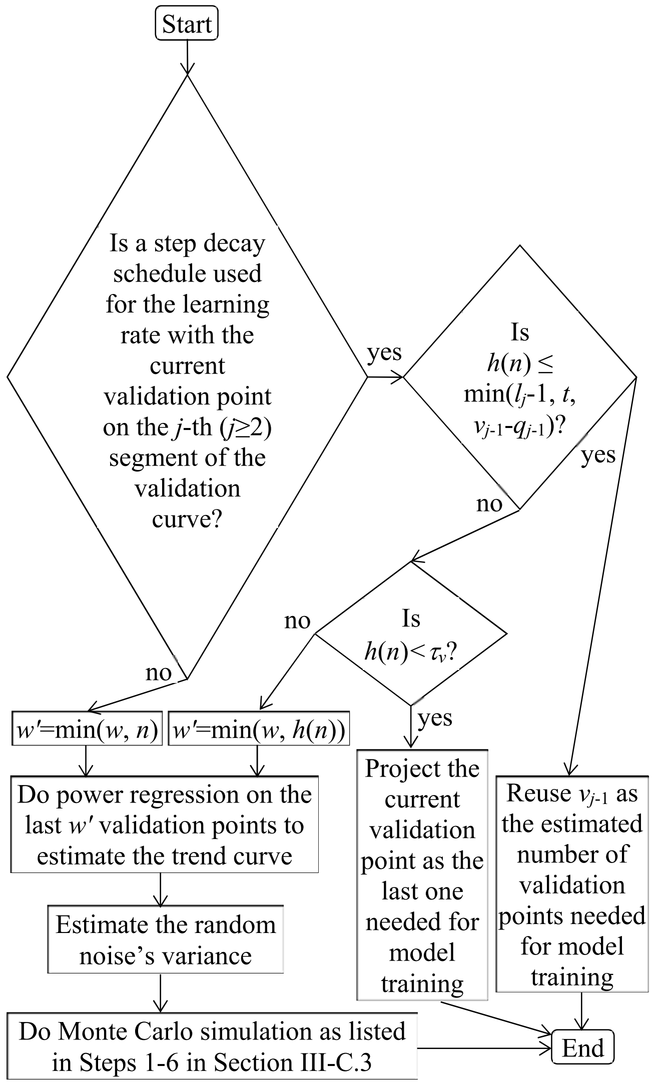

Start
Is a step decay schedule used for the learning rate with the current validation point on the j-th (j≥2) segment of the validation curve?
yes
no
Is h(n) ≤ min(l_j-1, t, v_j-1-q_j-1)?
yes
no
Is h(n)< τ_v?
no
yes
w′=min(w, n)
w′=min(w, h(n))
Do power regression on the last w′ validation points to estimate the trend curve
Estimate the random noise's variance
Do Monte Carlo simulation as listed in Steps 1-6 in Section III-C.3
Project the current validation point as the last one needed for model training
Reuse v_j-1 as the estimated number of validation points needed for model training
End

FIGURE 3. **The flow chart of our method for estimating the number of validation points needed for model training.**

C. ESTIMATING THE NUMBER OF VALIDATION POINTS NEEDED FOR MODEL TRAINING WHEN A FIXED LEARNING RATE IS USED DURING THE ENTIRE MODEL TRAINING PROCESS

This section focuses on the case where a fixed learning rate is used during the entire model training process. We show how to estimate the number of validation points needed for model training upon reaching a validation point that is after the τ_v-th one and at which the early stopping criterion is unmet. Section III-C.1 shows the regression method used to estimate the trend curve. Section III-C.2 covers how to estimate the random noise's variance. Section III-C.3 presents the Monte Carlo simulation approach used to project the number of validation points needed for model training.

1) ESTIMATING THE TREND CURVE

The validation error tends to decrease over time, whereas the rate of decrease typically reduces over time. In keeping with this, we use the same inverse power law function [7], [38] of the form

$$f(i)=ai^{-b}+c$$

as the regression function to model both the validation and trend curves (see Fig. 2). Here, i is the sequence number of the validation point, $a>0$, $b>0$, and $c>0$. We first fit the function to the validation curve up to the current validation point, and then use the fitted function to estimate the trend curve beyond that point.

Intuitively, the validation points well before the current one may not accurately reflect the validation curve's trend beyond the current validation point and could be unsuitable for function fitting. Thus, we use a pre-set window size w whose default value is 50 to skip these validation points. Let n denote the number of validation points obtained thus far. When fitting the regression function to the validation curve, we use the last

$$w'=\min(w, n)$$

validation points instead of all of the n validation points obtained thus far. To compute a, b, and c's values, we solve a constrained minimization problem:

$$\min \sum_{i=n-w'+1}^{n}[e_i-(ai^{-b}+c)]^2 \qquad (1)$$

the sum of the squared errors at the last w' validation points, subject to the constraints that $a>0$, $b>0$, and $c>0$. Recall that e_i is the validation error at the i-th validation point. One way to do constrained minimization is to use the truncated Newton method [39, Ch. 7.1] and initialize a, b, and c as one, one, and zero, respectively.

2) ESTIMATING THE RANDOM NOISE'S VARIANCE

Recall that we regard the validation curve as the sum of a smooth trend curve and some zero-mean random noise. e_i, $f(i)$, and e_i-$f(i)$ are the validation error, the estimated value of the trend curve, and the estimated value of the random noise at the i-th validation point, respectively. n is the number of validation points obtained so far. w' is the number of validation points used to fit the regression function. We use the last w' validation points to estimate the random noise's variance as

$$\hat{\sigma}^2=\frac{1}{w'}\sum_{i=n-w'+1}^{n}[e_i-f(i)]^2. \qquad (2)$$

3) PROJECTING THE NUMBER OF VALIDATION POINTS NEEDED FOR MODEL TRAINING

We use a Monte Carlo simulation method to project the number of validation points needed for model training. To the best of our knowledge, this is the first time Monte Carlo simulation has been used for progress indication. Our method works as follows:

1) Step 1: For each i ($n+1\leq i\leq v_{max}$), compute the estimated value $f(i)$ of the trend curve at the i-th validation point. Recall that n is the number of validation points obtained thus far. v_{max} is the maximal number of validation points allowed for model training. All of these $f(i)$ ($n+1\leq i\leq v_{max}$)

form the estimated trend curve beyond the current validation point, up to the last one allowed for model training.

2) Step 2: For each i ($n+1 \leq i \leq v_{max}$), randomly sample a number n_i from the normal distribution $N(0, \hat{\sigma}^2)$ as simulated random noise at the i-th validation point. Recall that $\hat{\sigma}^2$ is the estimated variance of the random noise. $f(i)+n_i$ is a simulated validation error at the i-th validation point. All of the $f(i)+n_i$ ($n+1 \leq i \leq v_{max}$) form a synthetic validation curve beyond the current validation point, up to the last one allowed for model training.
3) Step 3: Connect the actual validation curve up to the current validation point and the synthetic validation curve beyond that point to obtain a full synthetic validation curve, which goes from the first validation point to the last one allowed for model training.
4) Step 4: For each i ($n+1 \leq i \leq v_{max}$), check one by one whether the early stopping criterion is met on the full synthetic validation curve at the i-th validation point. If the early stopping criterion is not met anywhere, we obtain v_{max} as a simulated number of validation points needed for model training, and b_{max} as a simulated number of batches needed for model training. Recall that v_{max} and b_{max} are the maximum number of validation points and the maximum number of batches allowed for model training, respectively. Otherwise, if the early stopping criterion is met on the full synthetic validation curve for the first time at the j-th ($n+1 \leq j \leq v_{max}$) validation point, we obtain j as a simulated number of validation points needed for model training, and $j \times g$ as a simulated number of batches needed for model training. Recall that g is the number of batches of model training between two consecutive validation points.
5) Step 5: Repeat Steps 2-4 k times to obtain k simulated numbers of validation points needed for model training, which we term simulated estimates. k is a pre-set parameter. We choose 2,000 as its default value to obtain enough simulated estimates for our projection purpose without incurring excessive simulation overhead.

One could use the mode of the k simulated estimates as the projected number of validation points needed for model training. Compared to the mean, the mode is a more robust statistic in the presence of outliers [40]. Yet, using the mode directly is suboptimal. When there are ≥ 2 local modes with roughly the same frequency, which one of them is the global mode is somewhat random, resulting in instability of the projection. Considering this, we make a projection in the following way.

6) Step 6: By definition, every simulated estimate $\in [n+1, v_{max}]$. Divide $[n+1, v_{max}]$ into r disjoint intervals of equal width. r is a pre-set parameter whose default value is 200. Set a threshold
$$\gamma = k \times c_\gamma,$$
where c_γ is a coefficient whose default value is 0.04. Group the k simulated estimates by interval. Find every interval containing $>\gamma$ simulated estimates. Each such interval is regarded as a local mode. If the number of such intervals is ≥ 1, average the simulated estimates in all such intervals as the projected number of validation points needed for model training. Otherwise, if no such interval exists, the k simulated estimates spread relatively evenly across a wide range with no significant local mode. Their mean becomes the projected number of validation points needed for model training.

D. ESTIMATING THE NUMBER OF VALIDATION POINTS NEEDED FOR MODEL TRAINING WHEN A CONTINUOUS DECAY SCHEDULE FOR THE LEARNING RATE IS USED

This section focuses on the case where a continuous decay schedule for the learning rate is used. We show how to estimate the number of validation points needed for model training upon reaching a validation point that is after the τ_v-th one and at which the early stopping criterion is unmet.

In a continuous decay schedule, the learning rate shrinks continuously over epochs. For example, in an exponential decay schedule, the learning rate used in the i-th epoch is $\alpha e^{-i\rho}$ (see Fig. 4(a)). Here, $\alpha>0$ is the initial learning rate. $\rho>0$ is a constant controlling the learning rate's decay speed. Fig. 4(b) shows a typical validation curve in this case. The curve has roughly the same shape as an inverse power law function. Thus, we use the same method as that in Section III-C.1 to estimate the trend curve.

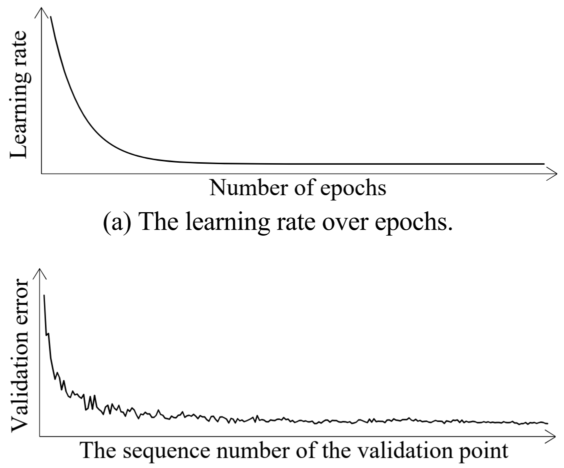


FIGURE 4. **The learning rate over epochs and a typical validation curve when an exponential decay schedule for the learning rate is used.**

To estimate the random noise's variance, we modify the method shown in Section III-C.2 in one respect. That method treats the random noise's variance as invariant over time. However, that is not the case with a continuous decay schedule, where the random noise's variance tends to shrink over time. The learning rate controls how much the neural network's weights and subsequently the validation error change both over time and due to random variation. The smaller the learning rate, the smaller the changes tend to be. If the learning rate=0, the neural network's weights and the validation error never differ from their initial values over time, and thus the random noise's variance=0.

Based on this intuition, we regard the random noise's standard deviation and variance as roughly proportional to the learning rate and its square, respectively. Let α_i and $\hat{\sigma}_i^2$ denote the learning rate and the estimated variance of the random noise at the i-th validation point, respectively. All of the α_i ($1 \leq i \leq v_{max}$) can be computed before model training begins. Recall that n is the number of validation points obtained thus far. To compute the estimated variance of the random noise $\hat{\sigma}_n^2$ at the current validation point, we still use the last w' validation points, but change formula (2) in Section III-C.2 to

$$\hat{\sigma}_n^2 = \frac{1}{w'} \sum_{i=n-w'+1}^{n} [\frac{e_i - f(i)}{\alpha_i/\alpha_n}]^2$$

to factor in the changes in variance over time. For each i ($n+1 \leq i \leq v_{max}$), we compute

$$\hat{\sigma}_i^2 = \hat{\sigma}_n^2 \times (\alpha_i/\alpha_n)^2.$$

To project the number of validation points needed for model training, we modify the method shown in Section III-C.3 at Step 2 alone. $\hat{\sigma}_i^2$ is the estimated variance of the random noise at the i-th validation point. For each i ($n+1 \leq i \leq v_{max}$), we randomly sample a number n_i from the normal distribution $N(0, \hat{\sigma}_i^2)$ instead of $N(0, \hat{\sigma}^2)$ as a simulated random noise at the i-th validation point.

E. ESTIMATING THE NUMBER OF VALIDATION POINTS NEEDED FOR MODEL TRAINING WHEN A STEP DECAY SCHEDULE FOR THE LEARNING RATE IS USED

This section focuses on the case where a step decay schedule for the learning rate is used. We show how to estimate the number of validation points needed for model training upon reaching a validation point that is after the τ_v-th one and at which the early stopping criterion is unmet.

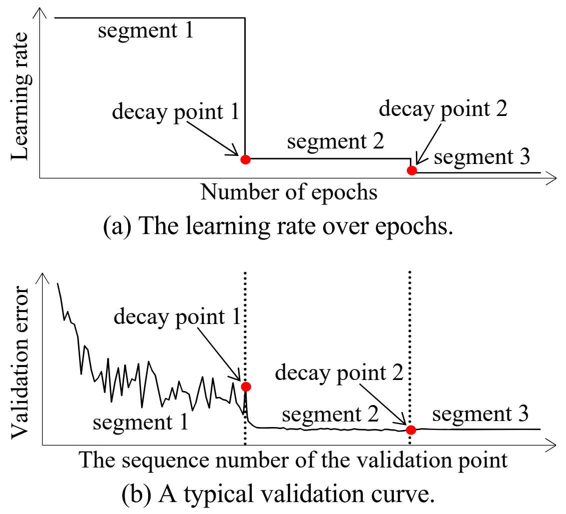


FIGURE 5. The learning rate over epochs and a typical validation curve when a step decay schedule for the learning rate is used.

In a step decay schedule, the learning rate is reduced by a fixed factor >1 after a certain number of epochs (see Fig. 5(a)). This number could vary over epochs in a pre-set way. Fig. 5(b) shows a typical validation curve in this case. We call each validation point where the learning rate is reduced a decay point. The decay points split the validation curve into multiple segments. For each $i \geq 1$, the i-th decay point is the first validation point on the (i+1)-th segment. Before model training starts, we can easily know the position of and the learning rate used on each segment.

We first consider the case that the current validation point is on the first segment of the validation curve. In this case, we use the method in Section III-C.1 to estimate the trend curve. We use the method in Section III-C.2 to compute $\hat{\sigma}_{(1)}^2$, the estimated variance of the random noise on the first segment. Let β_i denote the learning rate on the i-th segment, and $s(i)$ denote the sequence number of the segment where the i-th validation point is located. The learning rate at the i-th validation point is $\alpha_i = \beta_{s(i)}$. Recall that n is the number of validation points obtained thus far. v_{max} is the maximal number of validation points allowed for model training. For each i ($n+1 \leq i \leq v_{max}$), we compute the estimated variance of the random noise at the i-th validation point as

$$\begin{aligned}\hat{\sigma}_i^2 &= \hat{\sigma}_n^2 \times (\alpha_i/\alpha_n)^2 \\ &= \hat{\sigma}_{(1)}^2 \times (\beta_{s(i)}/\beta_1)^2.\end{aligned}$$

Then we use the method mentioned in the last paragraph of Section III-D to project the number of validation points needed for model training.

Next, we consider the case that the current validation point is on the j-th ($j \geq 2$) segment of the validation curve. As Fig. 5(b) shows, due to the learning rate reduction at a decay point, the validation curve often drops suddenly at both that point and the following several validation points. When we reach a validation point not far after such a decay point, if we use the method in Section III-C.1 to estimate the trend curve, this drop could cause the estimated trend curve to be inaccurate (see Fig. 6).

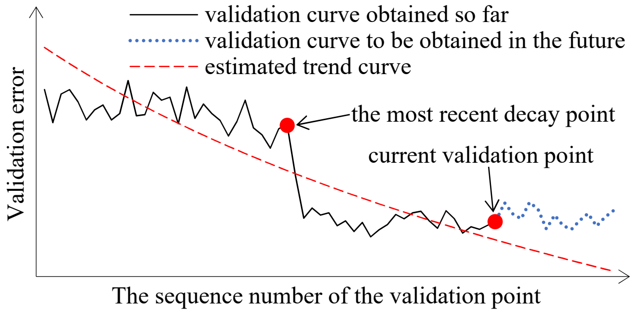


FIGURE 6. When reaching a validation point not far after the most recent decay point, using the method in Section III-C.1 to estimate the trend curve.

To address this issue, we modify the estimation methods in Sections III-C and III-D as follows. Let l_i denote the number of validation points on the i-th segment of the validation curve. Before model training starts, we can easily know each l_i.

$$q_{j-1} = \sum_{i=1}^{j-1} l_i$$

is the sequence number of the last validation point on the previous segment. Let v_{j-1} denote the number of validation points needed for model training that is estimated at the last validation point on the previous segment. If the estimated final validation point needed for model training is located on the current j-th segment, v_{j-1}-q_{j-1} is the sequence number of that validation point on the current j-th segment. Let $h(n)$ denote the sequence number of the current validation point on the current j-th segment. $h(n)$ is $\leq l_j$. We use a pre-set threshold t whose default value is 15 to identify the initial transient stage of each segment, where the validation error could change rapidly before becoming relatively stable on the later part of the segment. We differentiate between two possible sub-cases.

In the first sub-case, $h(n) \leq \min(l_j-1, t, v_{j-1}-q_{j-1})$. We posit the current validation point to be at the initial transient stage of the current segment of the validation curve. Since it is hard to use rapidly changing validation errors to obtain a good estimate of the number of validation points needed for model training, we reuse v_{j-1} as the estimate of this number. As t is small, we typically pass the initial transient stage in a relatively short amount of time.

In the second sub-case, $h(n) > \min(l_j-1, t, v_{j-1}-q_{j-1})$. We consider ourselves to have passed the initial transient stage and have reached the relatively stable stage of the current segment of the validation curve. Recall that to use the validation curve to refine the projected number of validation points needed for model training, we need at least τ_v validation points. τ_v's default value is three. Typically, $l_j-1 \geq \tau_v$ and $t \geq \tau_v$. If $h(n) < \tau_v$, which can occur if $v_{j-1}-q_{j-1} < \tau_v$, we project the current validation point as the last one needed for model training. Otherwise, if $h(n) \geq \tau_v$, we refine the projected number of validation points needed for model training in the following way. As Fig. 5(b) shows, if shifted to the left by q_{j-1} validation points, the current segment has roughly the same shape as an inverse power law function. Accordingly, we use the same shifted inverse power law function of the form

$$f(i)=a(i-q_{j-1})^{-b}+c$$

as the regression function to model the current segment of both the validation and trend curves. Recall that w is the maximum number of validation points allowed to fit the regression function. n is the number of validation points obtained thus far. $h(n)$ is the sequence number of the current validation point on the current segment. We use the last

$$w'=\min(w, h(n))$$

validation points on the current segment to fit the regression function to the validation curve, as well as to estimate the variance of the random noise at the current validation point as

$$\hat{\sigma}_n^2 = \frac{1}{w'} \sum_{i=n-w'+1}^{n} [e_i - f(i)]^2.$$

Recall that v_{max} is the maximal number of validation points allowed for model training. β_i is the learning rate on the i-th segment. $s(i)$ is the sequence number of the segment where the i-th validation point is located. The learning rate at the i-th validation point is $\alpha_i=\beta_{s(i)}$. For each i ($n+1 \leq i \leq v_{max}$), we compute the estimated variance of the random noise at the i-th validation point as

$$\hat{\sigma}_i^2 = \hat{\sigma}_n^2 \times (\alpha_i/\alpha_n)^2 = \hat{\sigma}_n^2 \times (\beta_{s(i)}/\beta_j)^2.$$

Then we use the method mentioned in the last paragraph of Section III-D to project the number of validation points needed for model training.

F. COMPLEXITY ANALYSIS FOR ESTIMATING THE NUMBER OF VALIDATION POINTS NEEDED FOR MODEL TRAINING

The key and most time-consuming step of our progress indication method is to estimate the number of validation points needed for model training. In this section, we discuss the worst-case computational complexity of this step, by defining each unit of computation as doing a basic (e.g., arithmetic) operation or computing an elementary (e.g., exponential or logarithmic) function.

As Fig. 3 shows, this estimation step typically involves three actions: 1) a power regression to estimate the trend curve, 2) estimating the random noise's variance, and 3) a Monte Carlo simulation to project the number of validation points needed for model training. When a step decay schedule is used for the learning rate and the current validation point is on the j-th ($j \geq 2$) segment of the validation curve, one of two exceptions could occur. First, if $h(n)$ is $\leq \min(l_j-1, t, v_{j-1}-q_{j-1})$, we reuse v_{j-1} as the estimated number of validation points needed for model training. Second, if $h(n)$ is $> \min(l_j-1, t, v_{j-1}-q_{j-1})$ and $h(n)$ is $< \tau_v$, we project the current validation point as the last one needed for model training. In the case of either exception, our estimation step has a computational complexity of $O(1)$.

In the rest of this section, we focus on the case that neither exception occurs. We first give the worst-case computational complexity of each of the three actions. Then we show the worst-case computational complexity of our estimation step.

1) THE WORST-CASE COMPUTATIONAL COMPLEXITY OF ACTION 1: DOING POWER REGRESSION TO ESTIMATE THE TREND CURVE

In this section, we give the worst-case computational complexity of using the truncated Newton method to do power regression to estimate the trend curve. As detailed in Nocedal and Wright [39, Ch. 7.1], to find the optimal point minimizing the objective function, this method starts from an initial point and uses a two-level nested loop to move the point towards the optimal point iteratively. The inner loop produces a search direction. In each round of the outer loop, the point is moved along the search direction.

Either loop could be terminated in one of two ways:

1) We preset a tolerance ε. The loop is terminated when a specific variable's value becomes $<\varepsilon$.
2) We preset ε and R_{max}, the maximum number of rounds allowed for the loop. The loop is terminated when a

specific variable's value becomes $<\varepsilon$ or the loop has run for R_{max} rounds, whichever occurs the first.

To the best of our knowledge, when the first way is used for both loops, the computational complexity of the truncated Newton method has not been given in any prior study nor can it be computed in an easy manner.

In the rest of this section, we focus on the second way that is often used in practice [41], [42]. Let $T_{max}\geq 1$ denote the maximum number of rounds allowed for the outer loop, and $I_{max}\geq 1$ denote the maximum number of rounds allowed for the inner loop in each round of the outer loop. In the worst case, the outer loop runs for T_{max} rounds, in each of which the inner loop runs for I_{max} rounds. From the description of the truncated Newton method in Nocedal and Wright [39, Ch. 7.1], we see that each round of the inner loop computes $O(1)$ Hessian-vector products, vector products, and gradients. Excluding the inner loop, the rest of each round of the outer loop computes $O(1)$ gradients. Putting these two parts together, each round of the outer loop computes

$$O(I_{max})+O(1)=O(I_{max})$$

Hessian-vector products, vector products, and gradients. Using the truncated Newton method to do power regression requires computing $O(T_{max}\times I_{max})$ Hessian-vector products, vector products, and gradients.

Computing a vector product involves $O(d)$ basic operations, where d is the number of input variables of the objective function. For our objective function shown in formula (1) with three input variables a, b, and c, d=3.

To calculate a Hessian-vector product or a gradient, we can use finite differencing [39, Ch. 8.1] that involves computing our objective function $O(d)$ times. Each such computation requires doing $O(w')$ basic operations and calculating $O(w')$ elementary functions. Recall w' is the number of validation points used to fit the regression function.

Putting everything together, calculating a Hessian-vector product or a gradient has a computational complexity of

$$O(d)\times O(w')=O(w').$$

The worst-case computational complexity of using the truncated Newton method to do power regression is

$$O(T_{max}\times I_{max})\times(O(d)+O(w'))$$
$$=O(T_{max}\times I_{max}\times w').$$

2) THE COMPUTATIONAL COMPLEXITY OF ACTION 2: ESTIMATING THE RANDOM NOISE'S VARIANCE

The last w' validation points are used to estimate the random noise's variance, with a computational complexity of $O(w')$.

3) THE WORST-CASE COMPUTATIONAL COMPLEXITY OF ACTION 3: DOING MONTE CARLO SIMULATION TO PROJECT THE NUMBER OF VALIDATION POINTS NEEDED FOR MODEL TRAINING

In this section, we give the worst-case computational complexity of doing Monte Carlo simulation to project the number of validation points needed for model training. As shown in Section III-C.3, this simulation is done in six steps. We compute each step's computational complexity and sum them to obtain the final result.

In Step 1, the estimated value $f(i)$ of the trend curve is computed at v_{max}-n validation points. Recall v_{max} is the maximum number of validation points allowed for model training. n is the number of validation points obtained thus far. Computing $f(i)$ at a single validation point has a computational complexity of $O(1)$. Thus, Step 1 has a computational complexity of $O(v_{max}-n)$.

In Step 2, we generate v_{max}-n random samples n_i ($n+1\leq i\leq v_{max}$) from a normal distribution and compute $f(i)+n_i$. Each sample can be obtained via the Box-Muller transform [43], which has a computational complexity of $O(1)$. Generating the v_{max}-n random samples n_i ($n+1\leq i\leq v_{max}$) has a computational complexity of $O(v_{max}-n)$. Computing $f(i)+n_i$ ($n+1\leq i\leq v_{max}$) has the same computational complexity. Putting these two parts together, Step 2 has a computational complexity of $O(v_{max}-n)$.

In Step 3, v_{max}-n points on the synthetic validation curve are connected with the actual validation curve to obtain the full synthetic validation curve. This has a computational complexity of $O(v_{max}-n)$.

In Step 4, for each i ($n+1\leq i\leq v_{max}$), we check one by one whether the early stopping criterion is met on the full synthetic validation curve at the i-th validation point. In the worst case, we go over all of these v_{max}-n points and find out the early stopping criterion is not met anywhere. Thus, Step 4 has a worst-case computational complexity of $O(v_{max}-n)$.

Summing Steps 2-4, we get a worst-case computational complexity of $O(v_{max}-n)$. In Step 5, Steps 2-4 are repeated k times, with a worst-case computational complexity of $O(k(v_{max}-n))$.

In Step 6, we divide $[n+1, v_{max}]$ into r disjoint intervals of equal width, group the k simulated estimates by interval, and compute the projected number of validation points needed for model training. This has a computational complexity of $O(r)+O(k)$.

Summing the six steps, we obtain the worst-case computational complexity of doing Monte Carlo simulation as

$$O(v_{max}-n)+O(k(v_{max}-n))+O(r)+O(k)$$
$$=O(\max\{k(v_{max}-n), r\}).$$

The above derivation uses the fact that k>1 and $v_{max}-n\geq 1$. If $n=v_{max}$, we are at the last validation point ever allowed for model training. Monte Carlo simulation is not needed there.

4) THE WORST-CASE COMPUTATIONAL COMPLEXITY OF OUR ESTIMATION STEP

Summing the three actions, we obtain the worst-case computational complexity of estimating the number of validation points needed for model training as

$$O(T_{max}\times I_{max}\times w')+O(w')+O(\max\{k(v_{max}-n), r\})$$
$$=O(\max\{T_{max}\times I_{max}\times w', k(v_{max}-n), r\}).$$

The above derivation uses the fact that $T_{max}\times I_{max}\times w'\geq w'$ because $T_{max}\geq 1$ and $I_{max}\geq 1$.

IV. PERFORMANCE

In this section, we report the performance results of our progress indicators for deep learning model training. We implemented our techniques given in Section III in TensorFlow Version 1.13.1. TensorFlow is a widely used open-source deep learning software package developed by Google [13]. In all of our tests, the progress indicators could provide useful estimates and revise them every 10 seconds with negligible overhead. We regard this as having fulfilled the three progress indication goals set out in our prior paper [7]: continuously revised estimates, acceptable pacing, and minimal overhead.

A. EXPERIMENT DESCRIPTION

We conducted the experiments by running TensorFlow on a Digital Storm workstation with one GeForce RTX 2080 Ti GPU, one eight-core Intel Core i7-9800X 3.8GHz central processing unit, 64GB memory, one 500GB solid-state drive, one 3TB SATA disk, and running the Ubuntu 18.04.02 operating system. All of the deep learning models were trained on the GPU.

We tested two popular deep learning models: GoogLeNet [44], a convolutional neural network, and the Gated Recurrent Unit (GRU) model in Purushotham *et al*. [45], a recurrent neural network. Except for the learning rate schedule, the number of training instances in each batch, and the maximum number of epochs allowed for model training, all of the hyper-parameters were set to their default values used in the two models' open source code [46], [47]. For each model, we tested three learning rate schedules: with a fixed learning rate, an exponential decay, and a step decay, respectively. For each model, we also tested four widely used optimization methods for deep learning model training: adaptive moment estimation (Adam) [48], classical stochastic gradient descent (SGD) [49], root mean square propagation (RMSprop) [50], and adaptive gradient (AdaGrad) [51]. We show the test results for Adam and RMSprop. The test results for SGD and AdaGrad are similar and put in the Appendix. For RMSprop, we show the test results when a fixed learning rate is used to train GoogLeNet. The test results for the other cases of RMSprop are similar and put in the Appendix. All of the other test results shown in Section IV are for the case using the Adam optimization method.

We used two well-known benchmark data sets: CIFAR-10 [52] and MIMIC-III [53] (Table I). Each data instance in CIFAR-10 is an image whose size is shown in Table I. We trained GoogLeNet on CIFAR-10, by splitting CIFAR-10 into a training set and a validation set in the same way as that in Krizhevsky [52]. We trained the GRU model on a subset of the MIMIC-III data set, which Purushotham *et al*. [45] termed the "Feature Set A, 48-h data," for the mortality prediction task. Each data instance in this subset is a sequence whose length is shown in Table I. This subset was split into a training set and a validation set in the same way as that in Purushotham *et al*. [45].

TABLE I
THE DATA SETS USED FOR TESTING OUR PROGRESS INDICATION METHOD

Name	# of data instances in the training set	# of data instances in the validation set	data instance size	# of classes
CIFAR-10	50,000	10,000	image size: 32×32	10
Feature Set A, 48-h data	19,146	6,382	sequence length: 48	2

We ran two kinds of tests:

1) **Unloaded system test**: The model was trained on an unloaded system.
2) **Workload interference test**: We began model training on an unloaded system. In the middle of model training, we started another model training task that competed with the first model training task for GPU resources.

For the unloaded system test, we report the test results for each combination of a learning rate schedule and a deep learning model. The only exception is that for the step decay schedule, we show the test results for training GoogLeNet. The test results for training the GRU model are similar and put in the Appendix. For the workload interference test, we present the test results of using a fixed learning rate to train GoogLeNet. The test results for the other cases of the workload interference test provide no extra information and are omitted.

In each test, the number of training instances in each batch was set to 128. The number of batches of model training between two consecutive validation points was set to 200 and 50 for GoogLeNet and the GRU model, respectively. The maximum number of epochs allowed for model training was set to 150. The initial learning rate was set to 0.001, regardless of which learning rate schedule was used. The patience p was set to 39, an integer randomly chosen from the range [5, 50]. The min_delta δ was set to 0.00207, a number randomly chosen from the range [0, 0.01].

B. ACCURACY MEASURE

We adopted the average estimation error used in Chaudhuri *et al.* [14] to measure the accuracy of the estimates provided by the progress indicator. As shown in Fig. 7, the average estimation error is defined as the ratio of two numbers. The numerator is the area of the region between a curve and a straight diagonal line. The curve gives the remaining model training time projected by the progress indicator over time. The straight diagonal line depicts the actual remaining model training time. The denominator is the area of the right triangle formed by the straight diagonal line, the x-axis, and the y-axis. The smaller the average estimation error, the more accurate the estimates provided by the progress indicator.

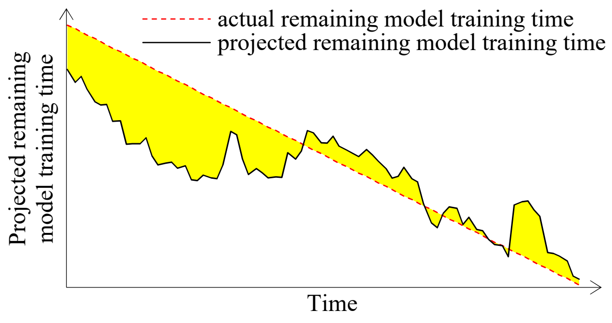


FIGURE 7. The numerator and denominator used to compute the average estimation error.

For each combination of a deep learning model, a learning rate schedule, a test type, and an optimization method, we trained the model five times, each in a separate run. We randomly choose one of these five runs and show the progress indicator's outputs over time in that run in Sections IV-C to IV-E and Sections A to C of the Appendix. In addition, we show the mean and the standard deviation of the average estimation error across the five runs in Section IV-F and Section D of the Appendix.

C. TEST RESULTS OF USING A FIXED LEARNING RATE DURING THE ENTIRE MODEL TRAINING PROCESS

In this section, we show the test results of using a fixed learning rate during the entire model training process.

1) UNLOADED SYSTEM TEST RESULTS FOR TRAINING GOOGLENET

In this test, GoogLeNet was trained on an unloaded system. The test's purpose is to show that when training GoogLeNet on an unloaded system using a fixed learning rate during the entire model training process, the progress indicator's estimates can be reasonably accurate for various optimization methods.

Using the Adam optimization method

We first consider the case that GoogLeNet was trained using the Adam optimization method. Fig. 8 shows the model training cost projected by the progress indicator over time, with the actual model training cost given by the horizontal dotted line. At the beginning of model training with no extra information, the progress indicator projected the model training cost based on the maximum number of validation points allowed for model training, which differed significantly from the actual number of validation points needed. Hence, the projected model training cost differed greatly from the actual model training cost. After reaching at least τ_v=3 validation points within 152 seconds, the progress indicator was able to revise the projected model training cost and make it more accurate.

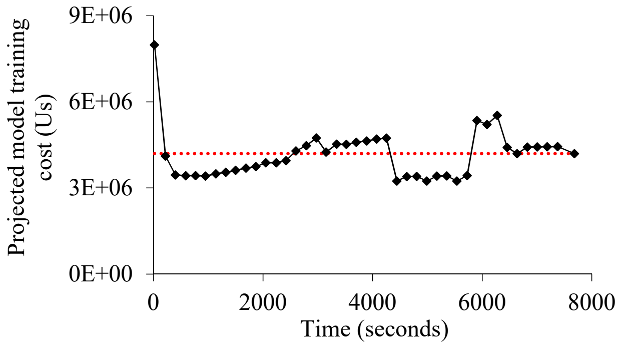


FIGURE 8. Model training cost projected over time (unloaded system test for training GoogLeNet using a fixed learning rate and Adam).

Fig. 9 shows the model training speed monitored by the progress indicator over time. During the entire model training process, the monitored model training speed was relatively stable.

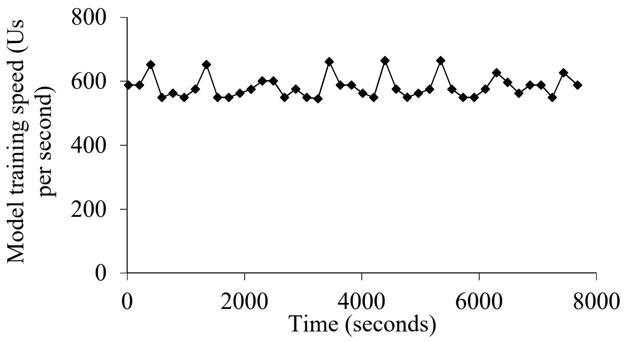


FIGURE 9. Model training speed over time (unloaded system test for training GoogLeNet using a fixed learning rate and Adam).

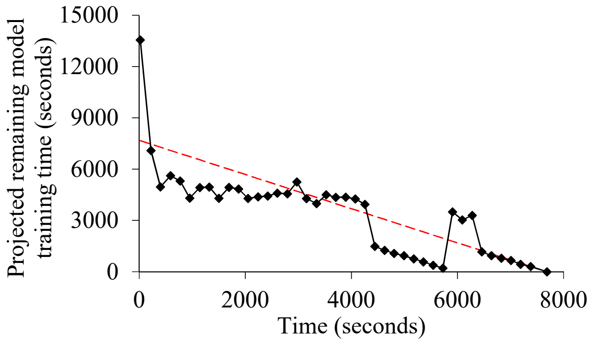


FIGURE 10. Remaining model training time projected over time (unloaded system test for training GoogLeNet using a fixed learning rate and Adam).

Fig. 10 shows the remaining model training time projected by the progress indicator over time, with the actual remaining model training time given by the dashed line. At the beginning of model training, the progress indicator's projected model training cost differed greatly from the actual model training cost. Thus, the remaining model training time projected by the progress indicator differed significantly from the actual one. Within 152 seconds, once the progress indicator was able to revise the projected model training cost and improve its accuracy, the projected remaining model training time became more precise.

Recall that the patience p was set to 39. By 4,392 seconds, the validation error had improved by $\leq$ the min_delta δ for eight validation points consecutively. From 4,392 to 5,690 seconds, the number of consecutive validation points for which the validation error had improved by $\leq\delta$ kept rising, causing the progress indicator to mistakenly project that model training could finish by ~5,800 seconds. Yet, the reality is that model training continued until 7,685 seconds. At 5,739 seconds, the validation error improved by $>\delta$, making the progress indicator realize that model training would take much longer than 5,800 seconds and revise its projections accordingly.

Fig. 11 shows the progress indicator's estimated percentage of model training work completed over time. Most of the time, the completed percentage curve is relatively close to the dotted diagonal line connecting the lower left corner and the upper right corner. A non-trivial deviation between the completed percentage curve and the diagonal line exists between 4,392 and 5,739 seconds for the reason given above.

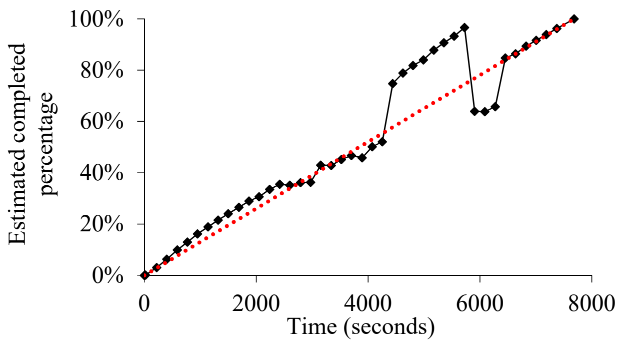


FIGURE 11. Completed percentage estimated over time (unloaded system test for training GoogLeNet using a fixed learning rate and Adam).

Using the RMSprop optimization method

Next, we consider the case that GoogLeNet was trained using the RMSprop optimization method. The performance results are shown in Fig. 12-15 and are similar to those shown in Fig. 8-11.

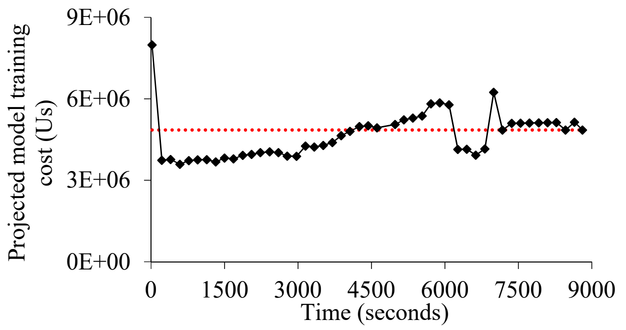


FIGURE 12. Model training cost projected over time (unloaded system test for training GoogLeNet using a fixed learning rate and RMSprop).

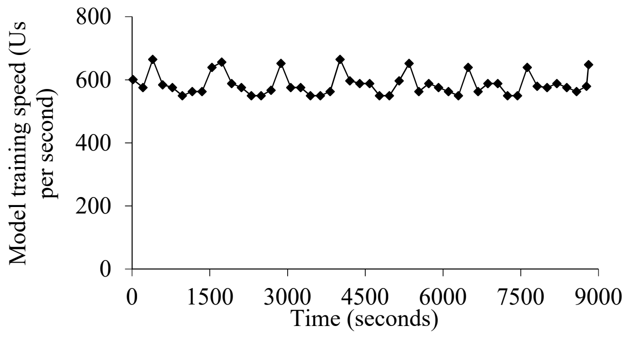


FIGURE 13. Model training speed over time (unloaded system test for training GoogLeNet using a fixed learning rate and RMSprop).

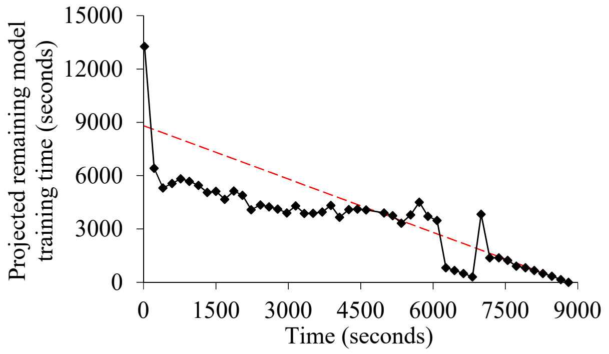


FIGURE 14. Remaining model training time projected over time (unloaded system test for training GoogLeNet using a fixed learning rate and RMSprop).

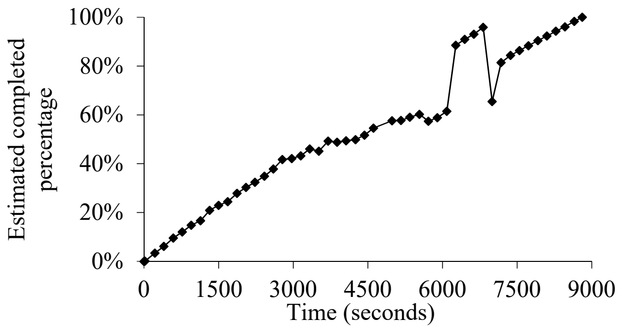


FIGURE 15. Completed percentage estimated over time (unloaded system test for training GoogLeNet using a fixed learning rate and RMSprop).

In the rest of Section IV, all of the test results shown were for training the deep learning model using the Adam optimization method.

2) WORKLOAD INTERFERENCE TEST RESULTS FOR TRAINING GOOGLENET

In the workload interference test, we began training GoogLeNet on an unloaded system using the same hyper-parameter values as those used in Section IV-C.1. In the middle of model training (at 3,600 seconds), we started another GoogLeNet training task that competed with the first model training task for GPU resources throughout the rest of the first task's execution. This extended the first task's running time. We present the performance results of the first task. This test's purpose is to show that our progress indicator can adjust to varying run-time system loads. In each figure of Section IV-C.2, we use a vertical dash-dotted line to give the point in time when the second GoogLeNet training task started running.

Fig. 16 shows the model training speed monitored by the progress indicator over time. Before the second model training task started running at 3,600 seconds, the shape of the curve in Fig. 16 is similar to that in Fig. 9. Once the second task started running, the monitored model training speed of the first task dropped roughly by half, as the second task was competing for GPU resources.

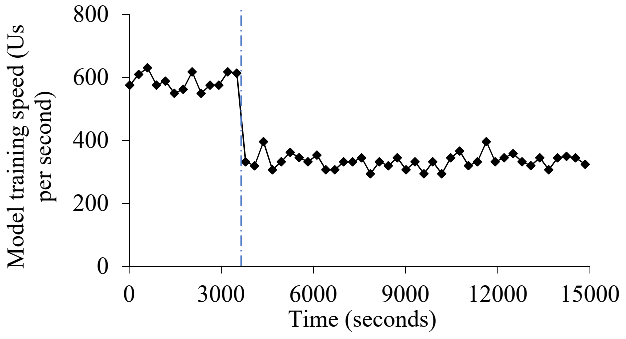


FIGURE 16. Model training speed over time (workload interference test for training GoogLeNet using a fixed learning rate and Adam).

Fig. 17 shows the remaining model training time projected by the progress indicator over time, with the actual remaining model training time given by the dashed line. Before the second model training task started running at 3,600 seconds, the shape of the curve in Fig. 17 is similar to that in Fig. 10. The progress indicator's projection error for the remaining model training time is mainly due to the unexpected large rise in system load starting from 3,600 seconds. After 3,600 seconds, the remaining model training time projected by the progress indicator became much more precise. The curve showing the projected remaining model training time becomes reasonably close to the dashed line.

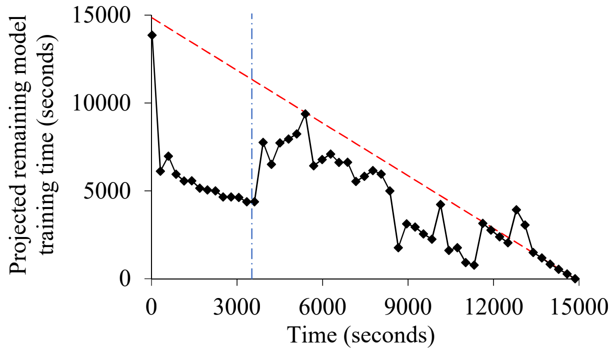


FIGURE 17. Remaining model training time projected over time (workload interference test for training GoogLeNet using a fixed learning rate and Adam).

Fig. 18 shows the progress indicator's estimated percentage of model training work completed over time. The estimated percentage tends to increase over time. The impact of running the second model training task is apparent starting at 3,600 seconds.

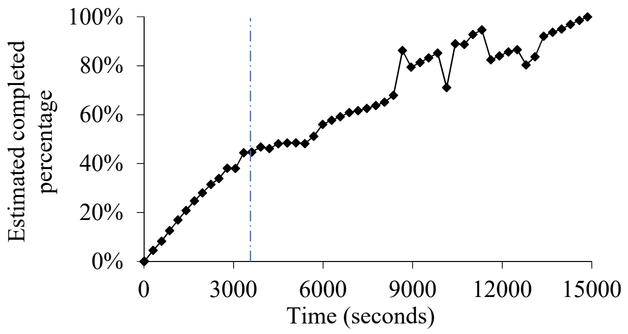


FIGURE 18. Completed percentage estimated over time (workload interference test for training GoogLeNet using a fixed learning rate and Adam).

3) UNLOADED SYSTEM TEST RESULTS FOR TRAINING THE GRU MODEL

In this test, the GRU model was trained on an unloaded system. The test's purpose is to show that the progress indicator's estimates can be reasonably accurate for different types of neural networks.

Fig. 19 shows the model training cost projected by the progress indicator over time, with the actual model training cost given by the horizontal dotted line. After reaching at least τ_v=3 validation points within 12 seconds, the progress indicator was able to project the model training cost reasonably accurately.

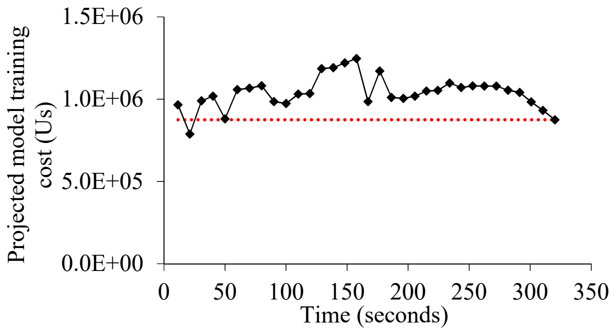


FIGURE 19. Model training cost projected over time (unloaded system test for training the GRU model using a fixed learning rate and Adam).

Fig. 20 shows the model training speed monitored by the progress indicator over time. Compared to that in Fig. 9, the curve in Fig. 20 is closer to a horizontal line, showing a more stable model training speed over time. The model training speed is computed based on the amount of work done in the past K=10 seconds. As mentioned in Section III-B, we regard the average amount of work needed for processing a validation instance one time to be U/3, as a rough approximation. Yet, this is not fully accurate, resulting in estimation errors on the amount of work done. In training the GRU model, it took about 3.5 seconds to go from one validation point to the next. As roughly the same number of validation instances were processed in each 10-second period, the estimation error of the amount of work completed is approximately the same across different 10-second periods, leading to a stable model training speed over time. By comparison, in training GoogLeNet, it took about 50 seconds to go from one validation point to the next one. The number of validation instances processed, and subsequently the estimation error of the amount of work completed, varies significantly across different 10-second periods, causing the monitored model training speed to vary more over time.

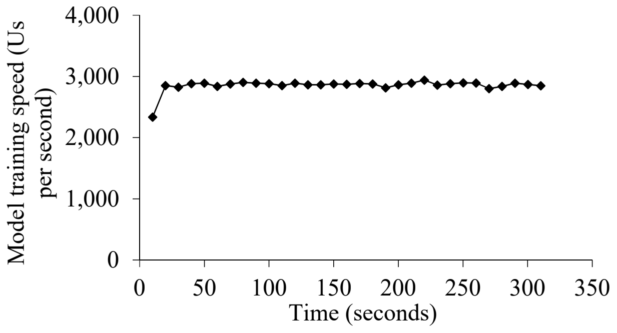


FIGURE 20. **Model training speed over time (unloaded system test for training the GRU model using a fixed learning rate and Adam).**

Fig. 21 shows the remaining model training time projected by the progress indicator over time, with the actual remaining model training time given by the dashed line. The projected remaining model training time is reasonably accurate.

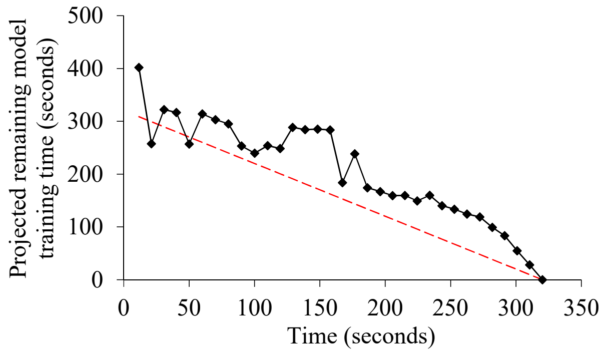


FIGURE 21. **Remaining model training time projected over time (unloaded system test for training the GRU model using a fixed learning rate and Adam).**

Fig. 22 shows the progress indicator's estimated percentage of model training work completed over time. The completed percentage curve is relatively close to the dotted diagonal line connecting the lower left corner and the upper right corner.

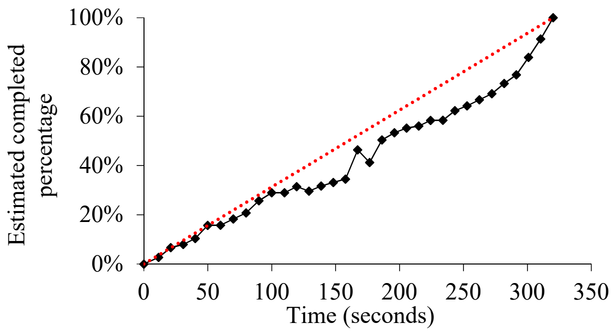


FIGURE 22. **Completed percentage estimated over time (unloaded system test for training the GRU model using a fixed learning rate and Adam).**

D. TEST RESULTS OF USING AN EXPONENTIAL DECAY SCHEDULE FOR THE LEARNING RATE

In this section, we show the test results of using an exponential decay schedule for the learning rate. Here, the constant ρ controlling the learning rate's decay speed was set to 0.05. The test's purpose is to show that the progress indicator's estimates can be reasonably accurate when an exponential decay schedule for the learning rate is used.

1) UNLOADED SYSTEM TEST RESULTS FOR TRAINING GOOGLENET

In this test, GoogLeNet was trained on an unloaded system using an exponential decay schedule for the learning rate. The performance results are shown in Fig. 23-26. From 0 to 3,945 seconds, the projected model training cost differed significantly from the actual one, leading to inaccurate projections of the remaining model training time and the percentage of model training work completed. Much of this inaccuracy results from the imprecise approximation we make in handling the exponential decay schedule, by treating the random noise's variance as roughly proportional to the square of the learning rate. After 3,945 seconds, the projections given by the progress indicator became much more accurate.

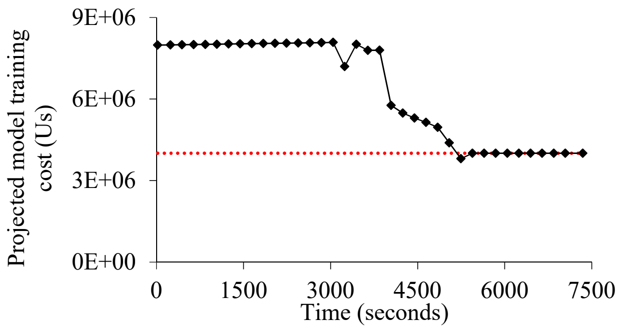


FIGURE 23. **Model training cost projected over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and Adam).**

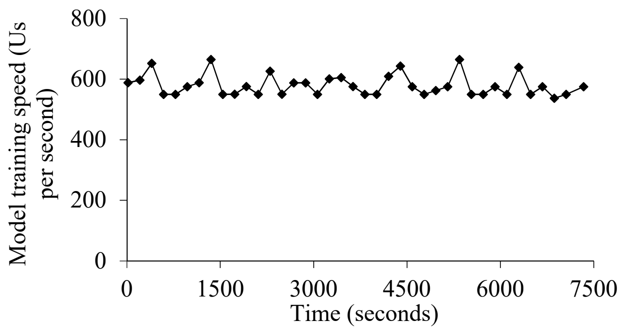


FIGURE 24. Model training speed over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and Adam).

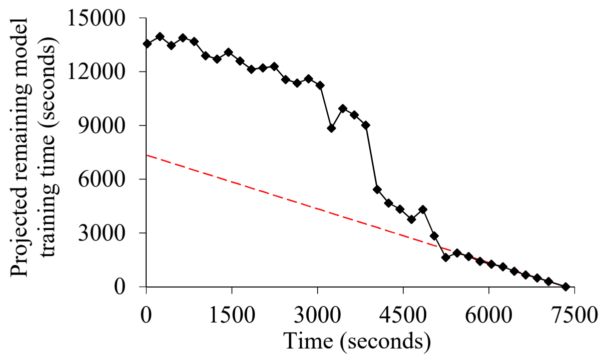


FIGURE 25. Remaining model training time projected over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and Adam).

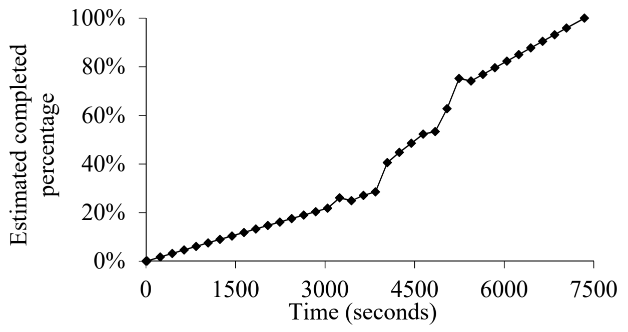


FIGURE 26. Completed percentage estimated over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and Adam).

2) UNLOADED SYSTEM TEST RESULTS FOR TRAINING THE GRU MODEL

In this test, the GRU model was trained on an unloaded system using an exponential decay schedule for the learning rate. The performance results are plotted in Fig. 27-30, showing the progress indicator made reasonably accurate projections.

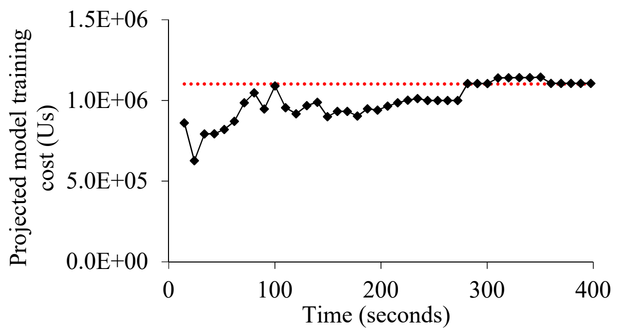


FIGURE 27. Model training cost projected over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and Adam).

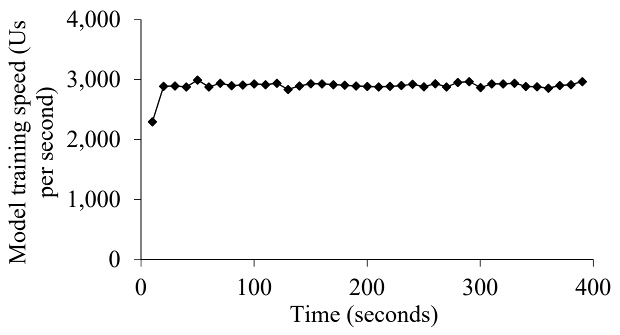


FIGURE 28. Model training speed over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and Adam).

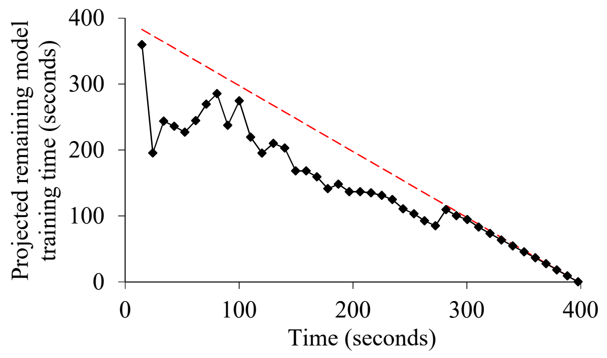


FIGURE 29. Remaining model training time projected over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and Adam).

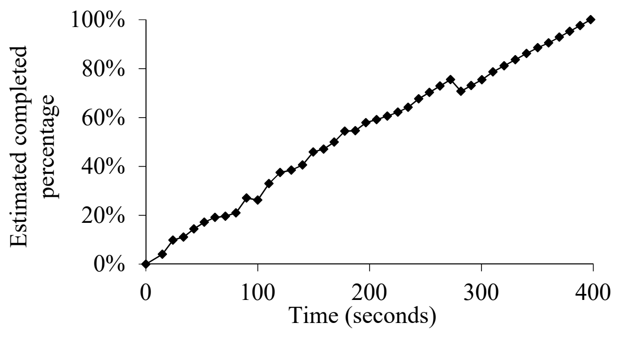


FIGURE 30. Completed percentage estimated over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and Adam).

E. UNLOADED SYSTEM TEST RESULTS FOR TRAINING GOOGLENET USING A STEP DECAY SCHEDULE FOR THE LEARNING RATE

In this section, we show the test results for training GoogLeNet on an unloaded system using a step decay schedule for the learning rate. Here, the learning rate was cut from 10^{-3} to 10^{-4} and 10^{-5} at the beginning of the 64-th and the 115-th epoch, respectively. The test's purpose is to show the progress indicator's estimates can be reasonably accurate when a step decay schedule for the learning rate is used.

In the test, early stopping occurred on the second segment of the validation curve, i.e., after the first decay point (see Fig. 5). The performance results are shown in Fig. 31-34 and similar to those shown in Fig. 8-11. In each figure of this section, we use a vertical dash-dotted line to give the time when the learning rate decay occurred.

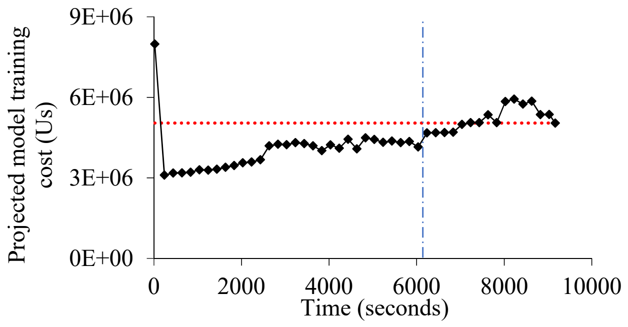


FIGURE 31. **Model training cost projected over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and Adam).**

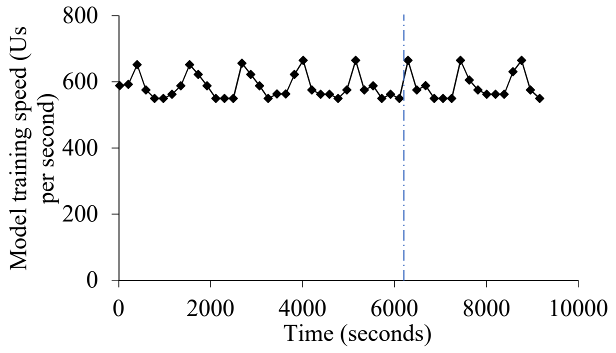


FIGURE 32. **Model training speed over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and Adam).**

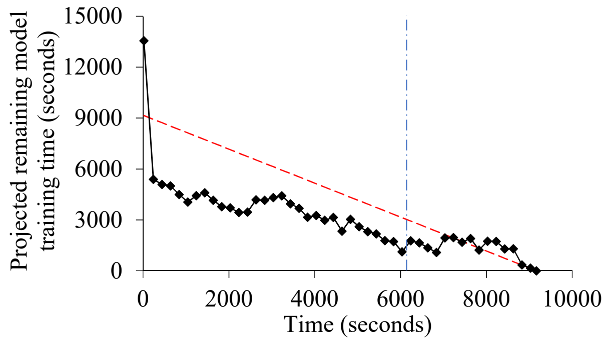


FIGURE 33. **Remaining model training time projected over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and Adam).**

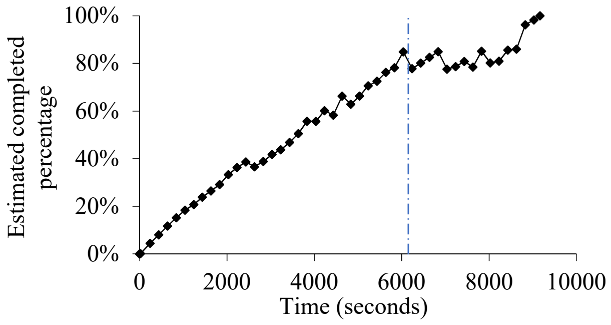


FIGURE 34. **Completed percentage estimated over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and Adam).**

F. SUMMARY STATISTICS OF THE AVERAGE ESTIMATION ERROR ACROSS THE FIVE RUNS

Recall that for each combination of a deep learning model, a learning rate schedule, a test type, and an optimization method, we trained the model five times, each in a separate run. For each combination presented in Sections IV-C to IV-E, we show the mean and the standard deviation of the average estimation error across the five runs in Table II. Except for one case, the average estimation error is ≤0.417 for every combination, indicating that our progress indicator offered reasonably accurate estimates of the remaining model training time.

TABLE II
FOR EACH COMBINATION OF A DEEP LEARNING MODEL, A LEARNING RATE SCHEDULE, A TEST TYPE, AND AN OPTIMIZATION METHOD PRESENTED IN SECTIONS IV-C TO IV-E, THE SUMMARY STATISTICS OF THE AVERAGE ESTIMATION ERROR ACROSS THE FIVE RUNS

Deep learning model	Learning rate schedule	Test type	Optimization method	Average estimation error
GoogLeNet	fixed learning rate	unloaded system test	Adam	0.272±0.060
	fixed learning rate	unloaded system test	RMSprop	0.417±0.140
	fixed learning rate	workload interference test	Adam	0.400±0.074
	exponential decay	unloaded system test	Adam	1.033±0.169
	step decay	unloaded system test	Adam	0.372±0.028
GRU	fixed learning rate	unloaded system test	Adam	0.362±0.034
	exponential decay	unloaded system test	Adam	0.303±0.074

G. SELECTING THE DEFAULT VALUES OF *w*, *r*, AND c_γ

Our progress indication method uses three key parameters: 1) *w*, the maximum number of validation points allowed to fit the regression function; 2) *r*, the number of disjoint intervals into which the possible range $[n+1, v_{max}]$ of the simulated number of validation points needed for model training is divided; and 3) c_γ, the coefficient used to compute the threshold γ. In this section, we show how we selected these parameters' default values by minimizing the mean of the average estimation errors of our progress indicator across several deep learning model training processes.

To do this selection, we used three popular deep learning models: VGG19 [54], a convolutional neural network, the long short-term memory (LSTM) model [55], a recurrent neural network, and the GRU model. We used three benchmark data sets: ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) [56], the Large Movie Review Dataset [57], and THUCNews [58] (Table III).

TABLE III
THE DATA SETS USED FOR SELECTING THE DEFAULT VALUES OF *w*, *r*, AND c_γ

Name	# of data instances in the training set	# of data instances in the validation set	data instance size	# of classes
Subset of ILSVRC2012	44,570	1,750	image size: 32×32	35
Large Movie Review Dataset	23,000	2,000	sequence length: 10-2,470 (234 on average)	2
Subset of THUCNews	50,000	5,000	sequence length: 8-26,849 (903 on average)	10

We trained VGG19 on a subset of ILSVRC2012. ILSVRC2012 has 1,000 image classes, ~1.2 million images intended for model training, and 50,000 images intended for model validation. We randomly chose 35 image classes and resized each image in them to 32×32 to make it suitable for training VGG19. All of the 44,570 images intended for model training in these classes were put into the training set. All of the 1,750 images intended for model validation in these classes were put into the validation set.

We trained the LSTM model on the Large Movie Review Dataset. Each data instance in this data set is a sequence. Table III shows the length distribution of all of these sequences. This data set includes 25,000 data instances that can be used for model training and validation. We randomly chose 2,000 of them to put into the validation set. The rest of them were put into the training set.

THUCNews has 14 classes and 740,000 data instances. Each data instance in this data set is a sequence. We trained the GRU model on a subset of THUCNews used in the model's open source code [59]. This subset includes ten classes, each with 5,000 data instances for training and 500 data instances for validation. Table III shows the length distribution of all of the sequences in this subset.

When training VGG19 and the LSTM model, the number of batches of model training between two consecutive validation points was set to 200. When training the GRU model, the number of batches of model training between two consecutive validation points was set to 10. For all of the three models, the patience *p* was set to 27, an integer randomly chosen from the range [5, 50]. The min_delta δ was set to 0.00443, a number randomly chosen from the range [0, 0.01]. Moreover, we used the same hyper-parameter values, learning rate schedule, and optimization method as those used in the three model's open source code [59-61]:

1) When training VGG19, the Adam optimization method and a step decay schedule for the learning rate were used. The initial learning rate was set to 10^{-3}. The learning rate was reduced to 10^{-4} and 10^{-5} at the beginning of the 50-th and the 70-th epoch, respectively. The number of training instances in each batch was set to 128. The maximum number of epochs allowed for model training was set to 100.
2) When training the LSTM model, the Adam optimization method and a fixed learning rate of 10^{-3} were used. The number of training instances in each batch was set to 24. The maximum number of epochs allowed for model training was set to 104.
3) When training the GRU model, the Adam optimization method and a fixed learning rate of 10^{-3} were used. The number of training instances in each batch was set to 128. The maximum number of epochs allowed for model training was set to 10.

On an unloaded system, we trained VGG19 five times, the LSTM model five times, and the GRU model five times, each in a separate run. We found for our progress indication method, the mean of the average estimation errors across the 15 runs was minimized when w=50, r=200, and c_γ=0.04. These values were chosen as the default values of *w*, *r*, and c_γ.

H. SENSITIVITY ANALYSIS OF *w*, *r*, AND c_γ

In this section, we use several experiments to evaluate the impact of *w*, *r*, and c_γ on the accuracy of the estimates provided by the progress indicator. In each experiment, we varied one parameter's value while keeping the other parameters' values constant. The mean of the average estimation errors across all runs of all of the unloaded system tests shown in Sections IV-C to IV-E and Sections A-C in the Appendix served as the accuracy measure for the estimates provided by the progress indicator.

w (the maximum number of validation points allowed to fit the regression function)

The first experiment concerns *w*, the maximum number of validation points allowed to fit the regression function. The default value of *w* is 50. We varied *w* from 3 to 90. Fig. 35 shows *w*'s impact on the mean of the average estimation errors. When w=10, 20, 30, 40, or 60, the accuracy measures are approximately the same as when w=50. When *w* is too small, not enough validation points are used to fit the regression function. When w is too large, many validation points that are too old to properly reflect the validation curve's future trend are used to fit the regression function. In either case, the fitted regression function may not reflect the validation curve's future trend well, degrading the accuracy of the estimates provided by the progress indicator. The safe range for w is between 10 and 60. If w is outside of this safe range, the accuracy of the estimates provided by the progress indicator will drop.

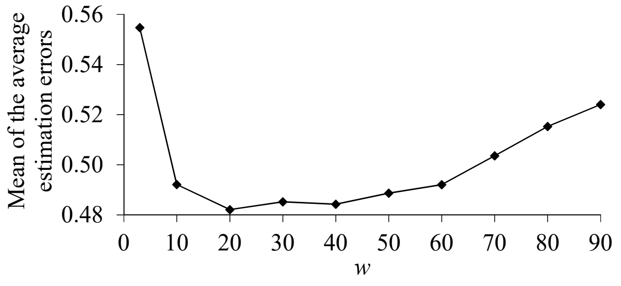


FIGURE 35. The mean of the average estimation errors vs. *w*.

r (the number of disjoint intervals into which the possible range of the simulated number of validation points needed for model training is divided)

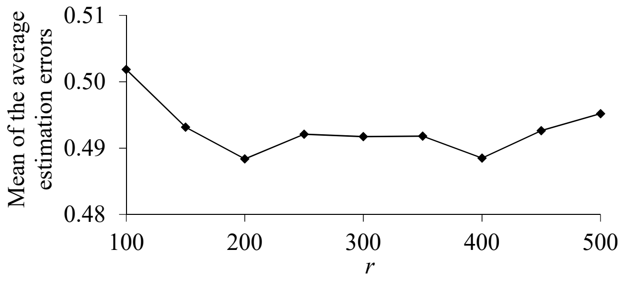


FIGURE 36. The mean of the average estimation errors vs. *r*.

The second experiment concerns r, the number of disjoint intervals into which the possible range $[n+1, v_{max}]$ of the simulated number of validation points needed for model training is divided (see Section III-C.3). The default value of r is 200. We varied r from 75 to 550. Fig. 36 shows r's impact on the mean of the average estimation errors. When r=150, 250, 300, 350, 400, or 450, the accuracy measures are approximately the same as when r=200. Recall that the projected number of validation points needed for model training is computed based on the intervals identified as local modes. When r is too small, each of the r divided intervals is large. An interval regarded as a local mode can contain much more than the actual local mode, introducing noise in estimating the number of validation points needed for model training. When r is too large, each of the r divided intervals is small. None of the r intervals may contain enough simulated numbers of validation points needed for model training and pass the threshold of being regarded as a local mode, even if some relevant local modes do indeed exist. In either case, the accuracy of the estimates provided by the progress indicator can degrade. The safe range for r is between 150 and 450.

c_γ (the coefficient used to compute γ)

The third experiment concerns c_γ, the coefficient used to compute the threshold γ. Recall that γ is used to decide whether an interval split from $[n+1, v_{max}]$ is a local mode or not. The projected number of validation points needed for model training is computed based on the identified local modes. The default value of c_γ is 0.04. We varied c_γ from 0.01 to 0.11. Fig. 37 shows c_γ's impact on the mean of the average estimation errors. When c_γ=0.03, 0.05, 0.06, or 0.07, the accuracy measures are roughly the same as when c_γ=0.04. When c_γ is too large, some relevant local modes may be excluded. When c_γ is too small, some intervals regarded as local modes may not be real local modes. In either case, the accuracy of the estimates provided by the progress indicator can degrade. The safe range for c_γ is between 0.03 and 0.07.

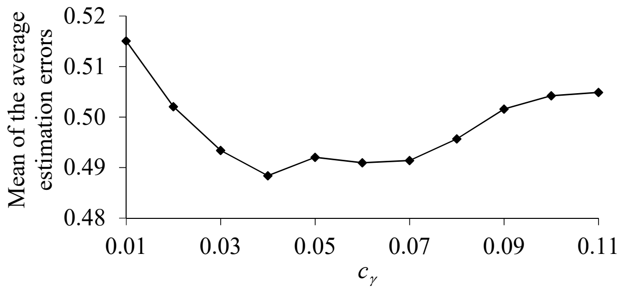


FIGURE 37. The mean of the average estimation errors vs. c_γ.

In summary, each of the parameters has a reasonably large safe range, within which the accuracy of the estimates provided by the progress indicator is insensitive to parameter value changes. For each parameter, its default value is within its safe range. If a parameter is outside of its safe range, the accuracy of the estimates provided by the progress indicator may drop.

V. DISCUSSION

This work focuses on developing system techniques to support progress indicators for deep learning model training. In this section, we describe several theoretical issues as being potentially interesting areas for future work.

When a fixed learning rate is used during the entire model training process, the method for estimating the number of validation points needed for model training in Section III-C treats the random noise's variance as invariant over time. Yet, in reality, as the validation error tends to decrease more slowly over time, the random noise's variance tends to reduce over time. Factoring this into our estimation method could improve its accuracy. One way to do this is to use a decay factor to model the reduction of the random noise's variance over time. Ideally, the decay factor should be derived based on a theoretical underpinning.

This work gives no upper bound on the progress indicator's estimation errors for the model training cost. It would be interesting to derive such upper bounds, possibly under certain conditions, similar to what Chaudhuri *et al*. [62] did for database query progress indicators.

This work uses only the data collected from the current model training process to estimate the regression function and the random noise's variance. In practice, a lot of data from the previous model building processes are often available. Meta-learning can be done on these data to improve the progress indicator's estimates for the current model training process. One way to do this is to compute weights based on the similarities of the validation curves from the previous model training processes and the current validation curve [63]. Then a weighted likelihood approach [64] is used to estimate the regression function and the random noise's variance for the current model training process.

In estimating the model training cost, as a rough approximation, we regard the cost of going backwards through the neural network once to be twice that of going forward through the neural network once. Yet, this is not fully accurate, resulting in estimation errors. To improve the estimation accuracy, we can develop more precise cost estimation models based on the type and architecture of the deep neural network and the activation functions used.

VI. CONCLUSIONS

In this paper, we present a detailed progress indicator implementation method for deep learning model training when early stopping is allowed. Our main idea is to use the validation curve to project the number of batches needed for model training. During model training, we keep refining the projected model training cost and checking the current model training speed. Periodically, we revise the projected fraction of model training work completed and the projected remaining model training time displayed to the user. Our experiments show that the resulting progress indicator can offer useful information even if the run-time system load varies over time. In addition, the progress indicator can self-correct its initial estimation errors, if any, over time. This demonstrates for the first time the feasibility of providing non-trivial progress indicators for deep learning model training when early stopping is allowed.

APPENDIX

In the appendix, we show the performance results not included in Section IV.

A. ADDITIONAL TEST RESULTS OF USING A FIXED LEARNING RATE DURING THE ENTIRE MODEL TRAINING PROCESS

1) UNLOADED SYSTEM TEST RESULTS FOR TRAINING GOOGLENET

Using the SGD optimization method

In this test, GoogLeNet was trained on an unloaded system using the SGD optimization method and a fixed learning rate during the entire model training process. The performance results are shown in Fig. 38-41. The early stopping criterion was never satisfied during the whole model training process. The progress indicator figured this out correctly and made accurate projections.

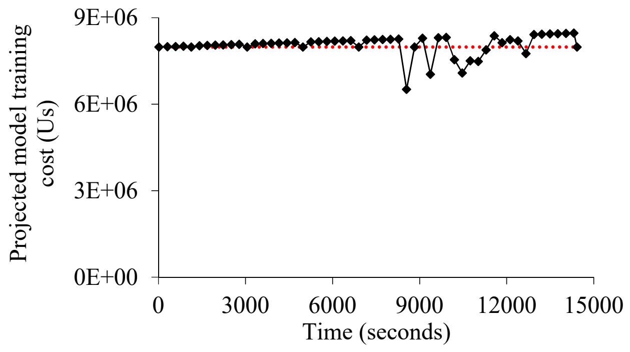


FIGURE 38. Model training cost projected over time (unloaded system test for training GoogLeNet using a fixed learning rate and SGD).

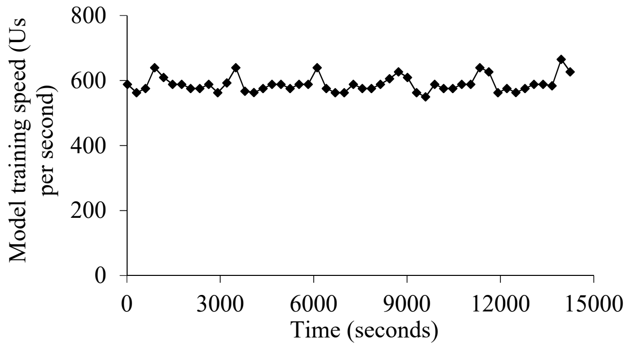


FIGURE 39. Model training speed over time (unloaded system test for training GoogLeNet using a fixed learning rate and SGD).

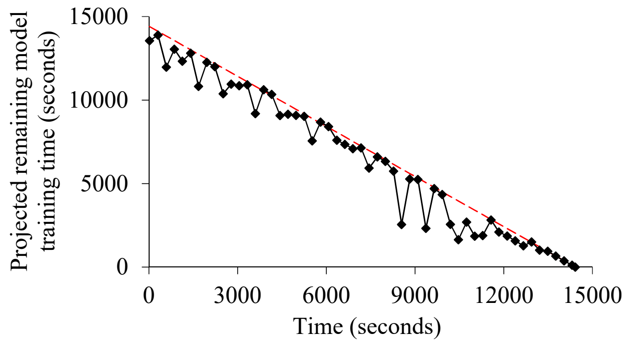


FIGURE 40. Remaining model training time projected over time (unloaded system test for training GoogLeNet using a fixed learning rate and SGD).

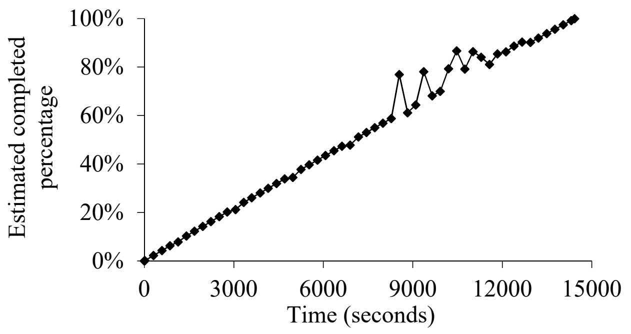


FIGURE 41. Completed percentage estimated over time (unloaded system test for training GoogLeNet using a fixed learning rate and SGD).

Using the AdaGrad optimization method

In this test, GoogLeNet was trained on an unloaded system using the AdaGrad optimization method and a fixed learning rate during the entire model training process. The performance results are shown in Fig. 42-45 and are similar to those shown in Fig. 38-41.

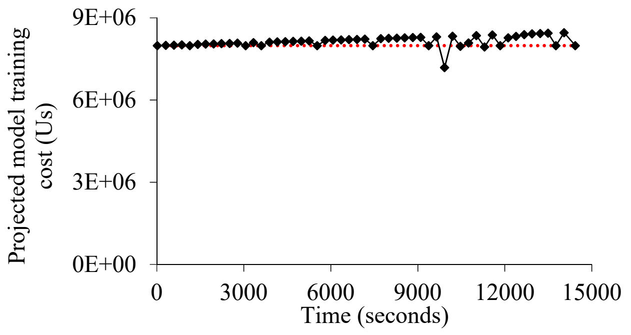


FIGURE 42. Model training cost projected over time (unloaded system test for training GoogLeNet using a fixed learning rate and AdaGrad).

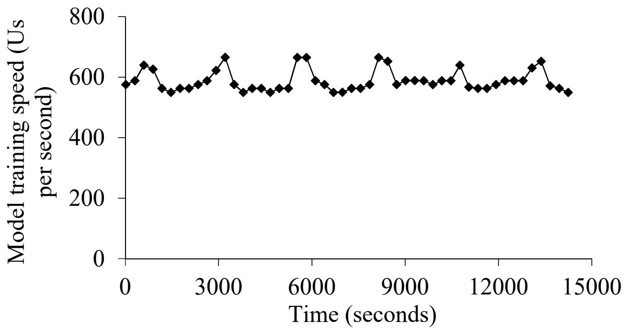


FIGURE 43. Model training speed over time (unloaded system test for training GoogLeNet using a fixed learning rate and AdaGrad).

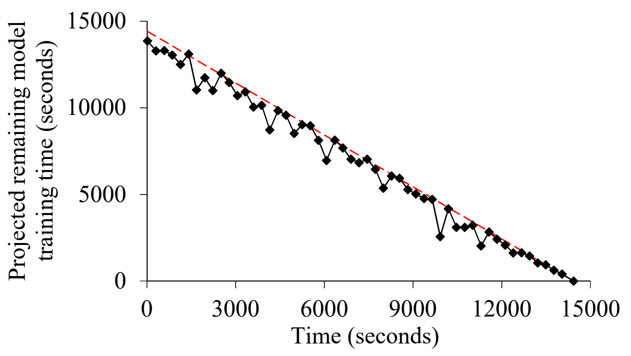


FIGURE 44. Remaining model training time projected over time (unloaded system test for training GoogLeNet using a fixed learning rate and AdaGrad).

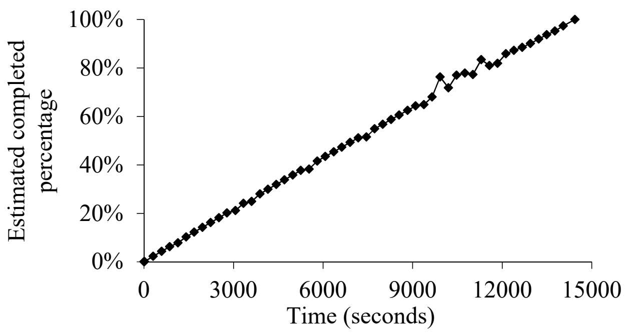


FIGURE 45. Completed percentage estimated over time (unloaded system test for training GoogLeNet using a fixed learning rate and AdaGrad).

2) UNLOADED SYSTEM TEST RESULTS FOR TRAINING THE GRU MODEL

Using the RMSprop optimization method

In this test, the GRU model was trained on an unloaded system using the RMSprop optimization method and a fixed learning rate during the entire model training process. The performance results are shown in Fig. 46-49 and are similar to those shown in Fig. 19-22.

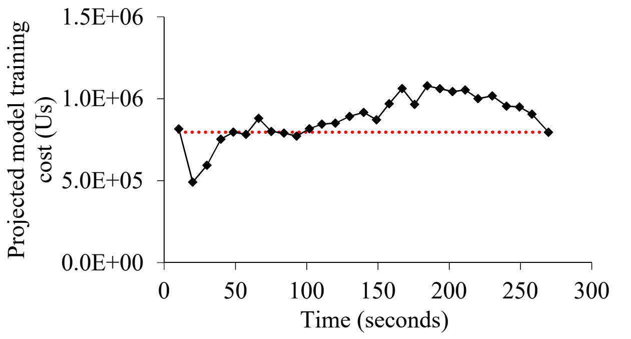


FIGURE 46. Model training cost projected over time (unloaded system test for training the GRU model using a fixed learning rate and RMSprop).

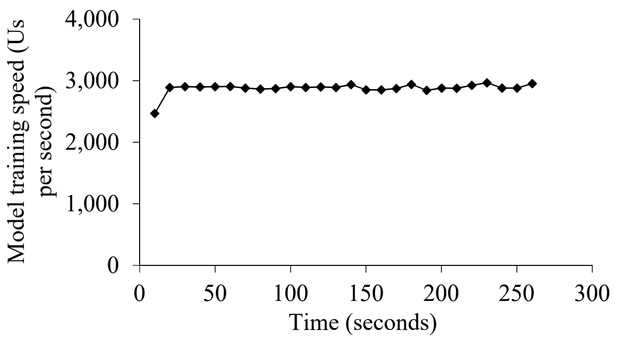


FIGURE 47. Model training speed over time (unloaded system test for training the GRU model using a fixed learning rate and RMSprop).

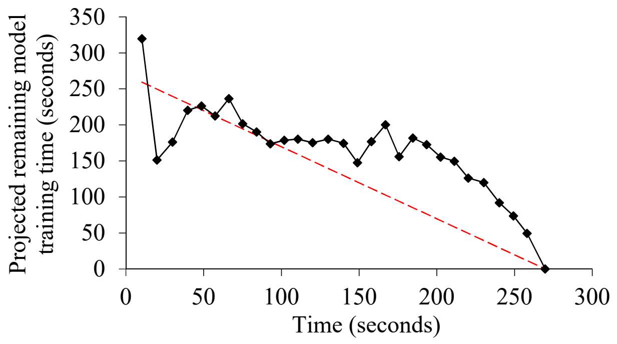


FIGURE 48. Remaining model training time projected over time (unloaded system test for training the GRU model using a fixed learning rate and RMSprop).

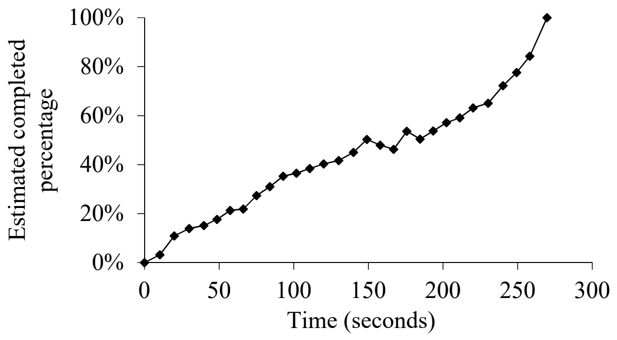


FIGURE 49. Completed percentage estimated over time (unloaded system test for training the GRU model using a fixed learning rate and RMSprop).

Using the SGD optimization method

In this test, the GRU model was trained on an unloaded system using the SGD optimization method and a fixed learning rate during the entire model training process. The performance results are shown in Fig. 50-53. From 163 to 388 seconds, the projected model training cost differed significantly from the actual one, leading to inaccurate projections of the remaining model training time and the percentage of model training work completed. Much of this inaccuracy results from power regression's inability to accurately estimate the trend curve during this time period.

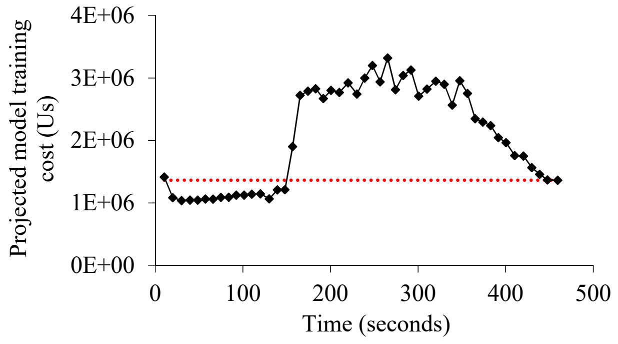


FIGURE 50. Model training cost projected over time (unloaded system test for training the GRU model using a fixed learning rate and SGD).

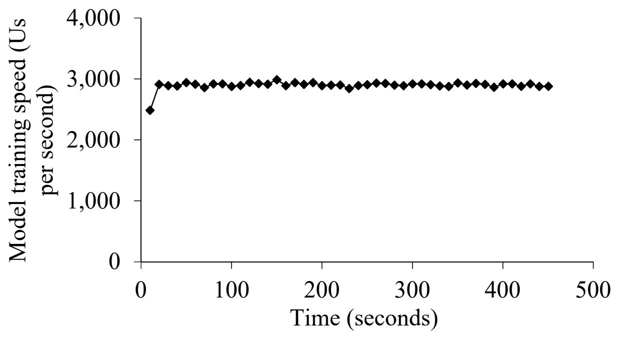


FIGURE 51. Model training speed over time (unloaded system test for training the GRU model using a fixed learning rate and SGD).

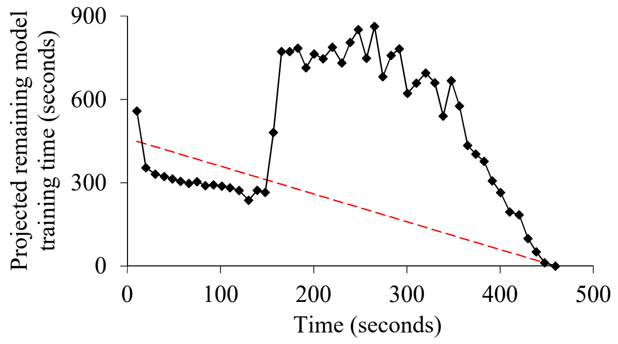


FIGURE 52. Remaining model training time projected over time (unloaded system test for training the GRU model using a fixed learning rate and SGD).

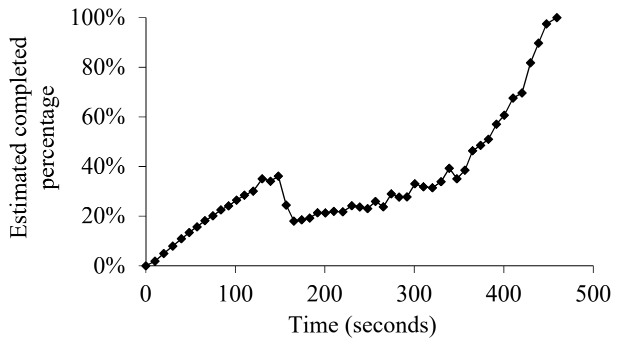


FIGURE 53. Completed percentage estimated over time (unloaded system test for training the GRU model using a fixed learning rate and SGD).

Using the AdaGrad optimization method

In this test, the GRU model was trained on an unloaded system using the AdaGrad optimization method and a fixed learning rate during the entire model training process. The performance results are shown in Fig. 54-57. At 10 seconds, only three validation points were available, making it difficult to estimate the trend curve accurately. Hence, the progress indicator made inaccurate projections. After 20 seconds, the projections given by the progress indicator became much more accurate as more validation points became available.

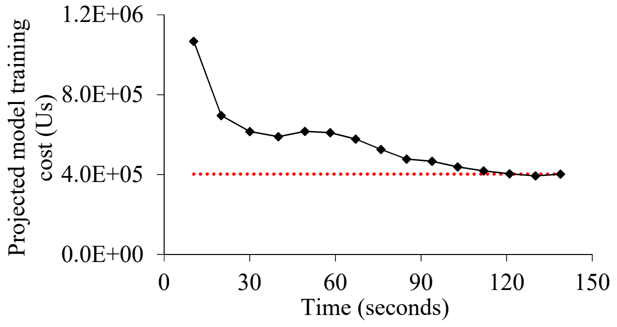


FIGURE 54. Model training cost projected over time (unloaded system test for training the GRU model using a fixed learning rate and AdaGrad).

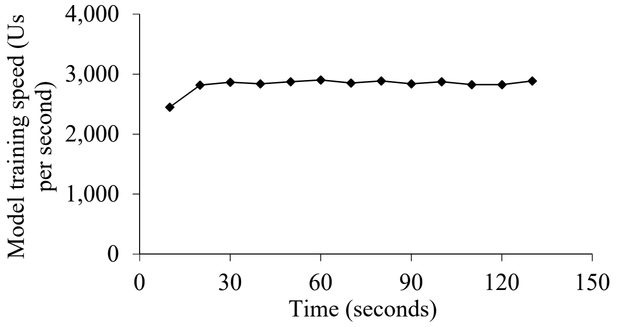


FIGURE 55. Model training speed over time (unloaded system test for training the GRU model using a fixed learning rate and AdaGrad).

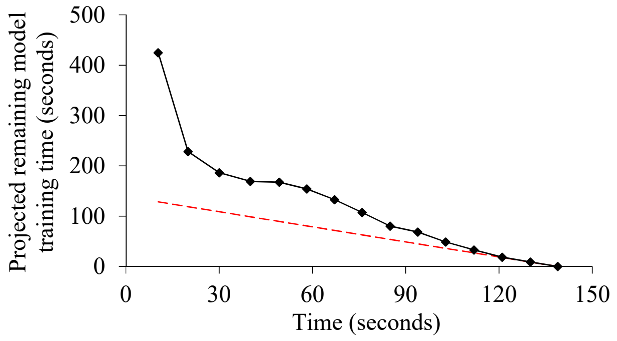


FIGURE 56. Remaining model training time projected over time (unloaded system test for training the GRU model using a fixed learning and AdaGrad).

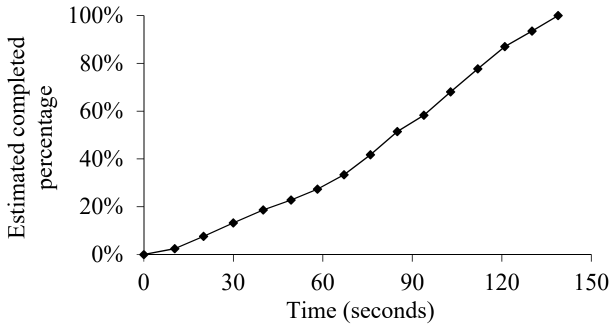


FIGURE 57. Completed percentage estimated over time (unloaded system test for training the GRU model using a fixed learning rate and AdaGrad).

B. ADDITIONAL TEST RESULTS OF USING AN EXPONENTIAL DECAY SCHEDULE FOR THE LEARNING RATE

1) UNLOADED SYSTEM TEST RESULTS FOR TRAINING GOOGLENET

Using the RMSprop optimization method

In this test, GoogLeNet was trained on an unloaded system using an exponential decay schedule for the learning rate and the RMSprop optimization method. The performance results are shown in Fig. 58-61 and are similar to those shown in Fig. 23-26.

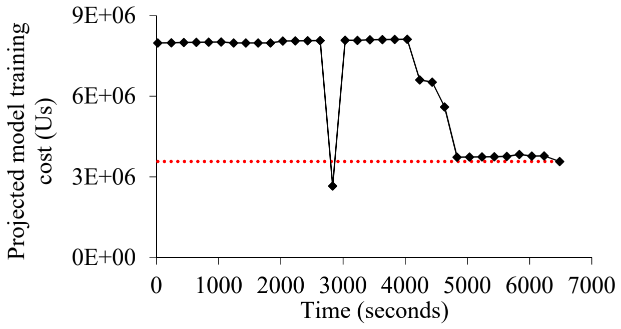


FIGURE 58. Model training cost projected over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and RMSprop).

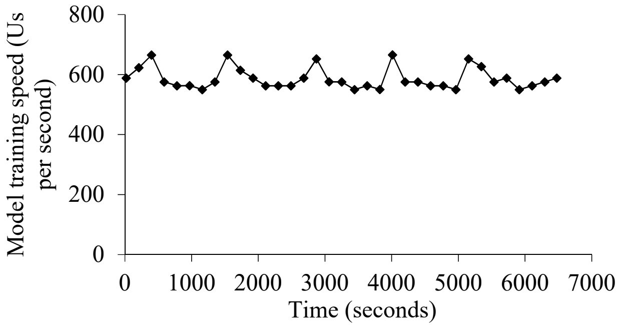


FIGURE 59. Model training speed over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and RMSprop).

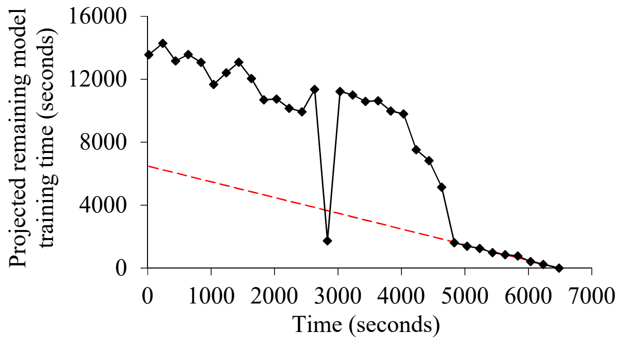


FIGURE 60. Remaining model training time projected over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and RMSprop).

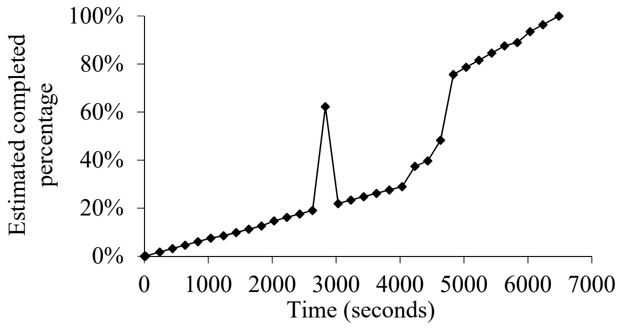


FIGURE 61. Completed percentage estimated over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and RMSprop).

Using the SGD optimization method

In this test, GoogLeNet was trained on an unloaded system using an exponential decay schedule for the learning rate and the SGD optimization method. The performance results are shown in Fig. 62-65 and are similar to those shown in Fig. 23-26.

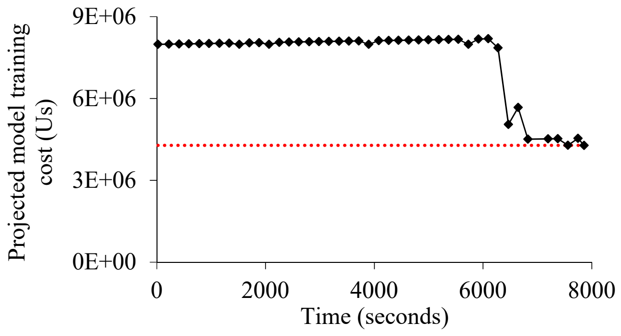


FIGURE 62. Model training cost projected over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and SGD).

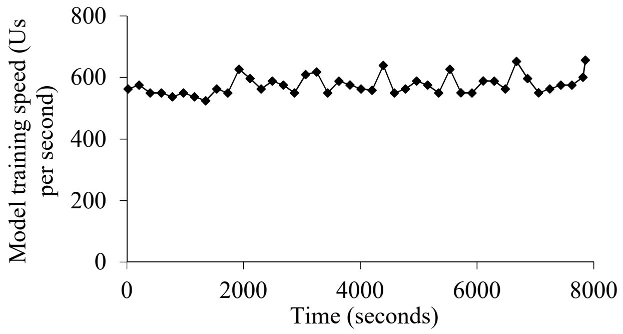


FIGURE 63. Model training speed over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and SGD).

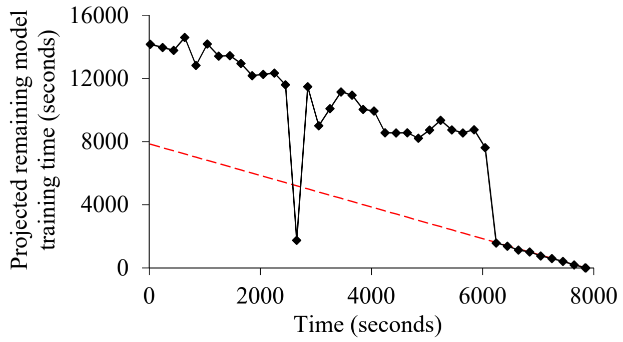


FIGURE 64. Remaining model training time projected over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and SGD).

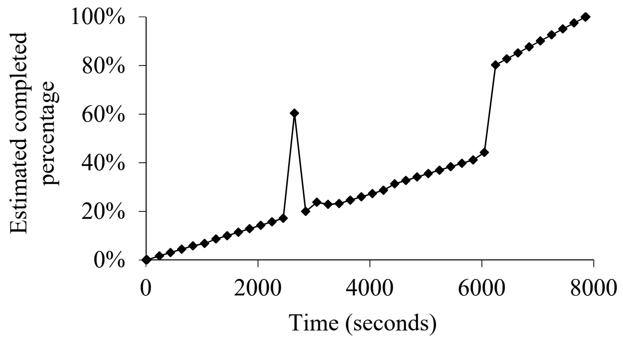


FIGURE 65. Completed percentage estimated over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and SGD).

Using the AdaGrad optimization method

In this test, GoogLeNet was trained on an unloaded system using an exponential decay schedule for the learning rate and the AdaGrad optimization method. The performance results are shown in Fig. 66-69 and are similar to those shown in Fig. 23-26.

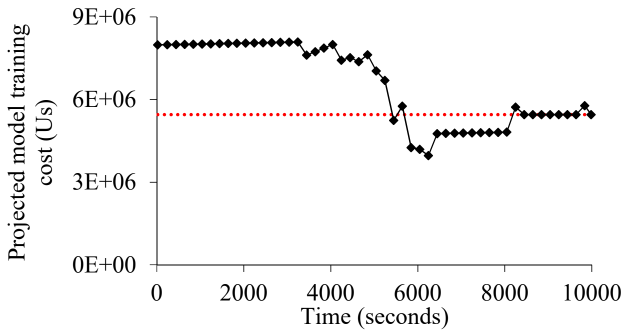


FIGURE 66. Model training cost projected over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and AdaGrad).

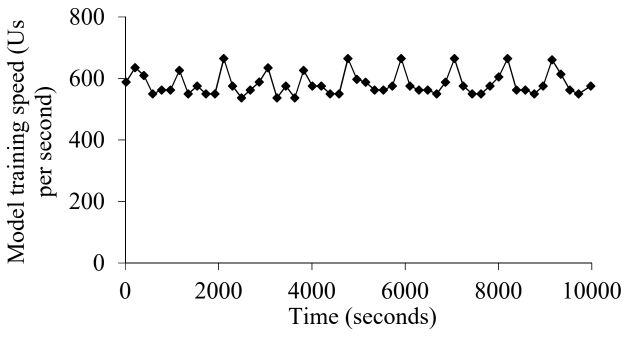


FIGURE 67. Model training speed over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and AdaGrad).

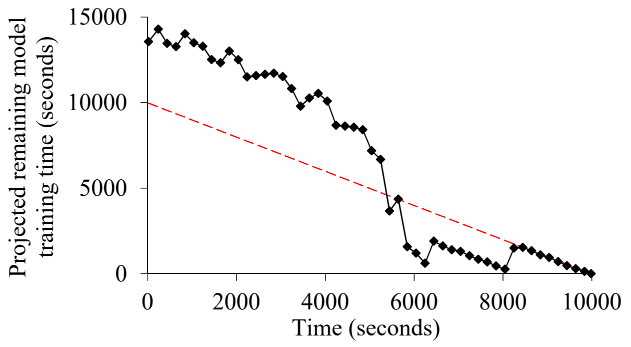


FIGURE 68. Remaining model training time projected over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and AdaGrad).

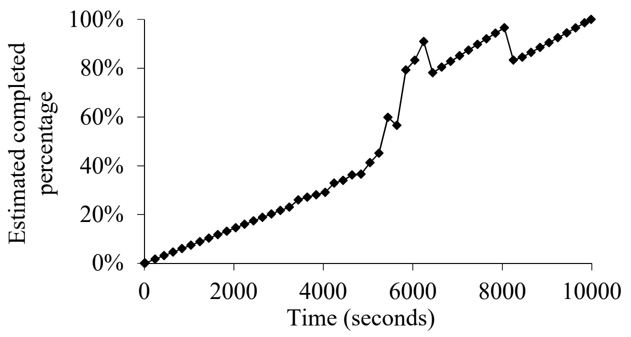


FIGURE 69. Completed percentage estimated over time (unloaded system test for training GoogLeNet using an exponential decay schedule for the learning rate and AdaGrad).

2) UNLOADED SYSTEM TEST RESULTS FOR TRAINING THE GRU MODEL

Using the RMSprop optimization method

In this test, the GRU model was trained on an unloaded system using an exponential decay schedule for the learning rate and the RMSprop optimization method. The performance results are shown in Fig. 70-73 and are similar to those shown in Fig. 27-30.

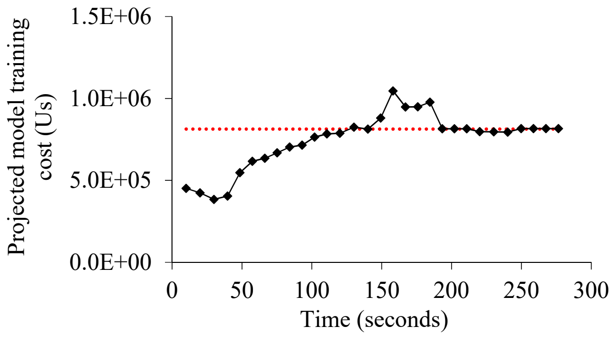


FIGURE 70. Model training cost projected over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and RMSprop).

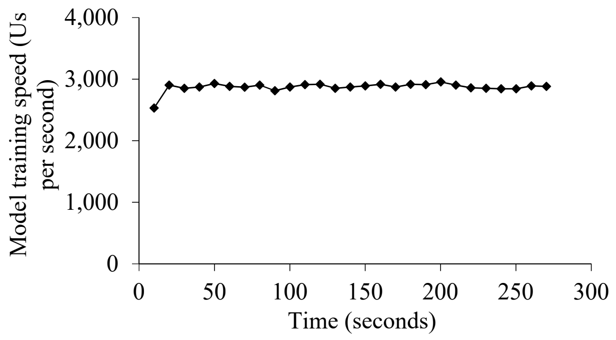


FIGURE 71. Model training speed over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and RMSprop).

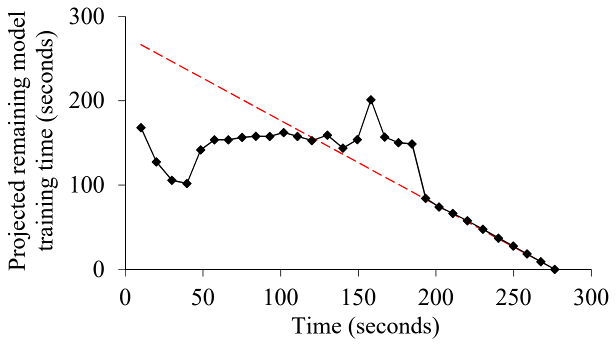


FIGURE 72. Remaining model training time projected over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and RMSprop).

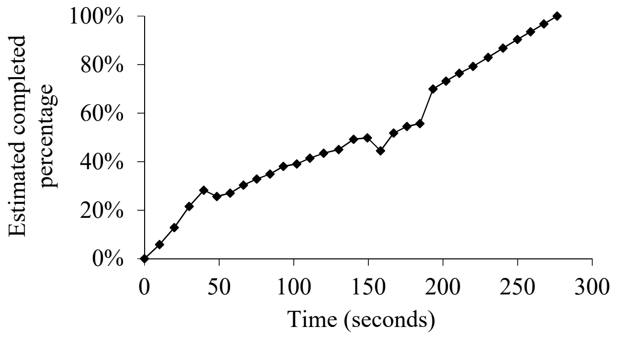


FIGURE 73. Completed percentage estimated over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and RMSprop).

Using the SGD optimization method

In this test, the GRU model was trained on an unloaded system using an exponential decay schedule for the learning rate and the SGD optimization method. The performance results are shown in Fig. 74-77 and are similar to those shown in Fig. 23-26.

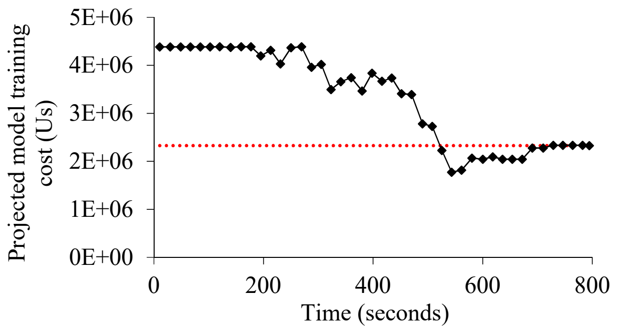


FIGURE 74. Model training cost projected over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and SGD).

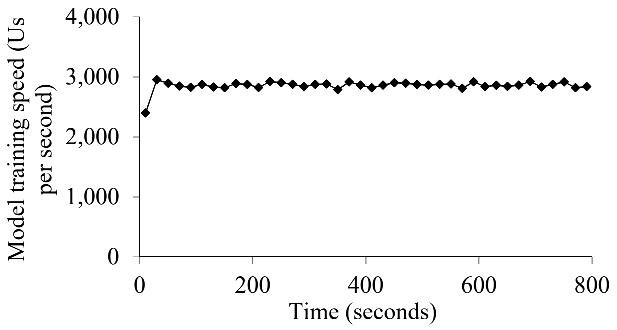


FIGURE 75. Model training speed over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and SGD).

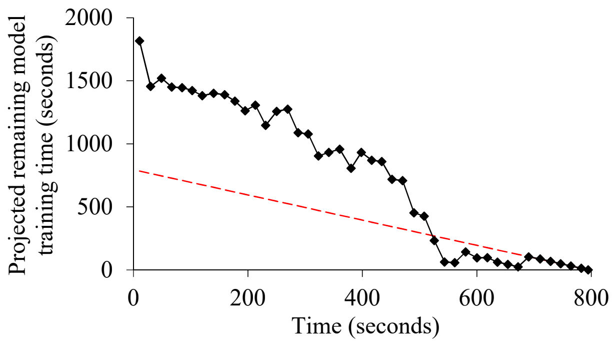


FIGURE 76. Remaining model training time projected over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and SGD).

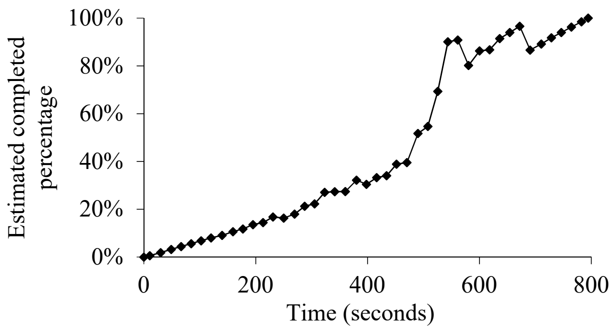


FIGURE 77. Completed percentage estimated over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and SGD).

Using the AdaGrad optimization method

In this test, the GRU model was trained on an unloaded system using an exponential decay schedule for the learning rate and the AdaGrad optimization method. The performance results are plotted in Fig. 78-81, showing the progress indicator made reasonably accurate projections.

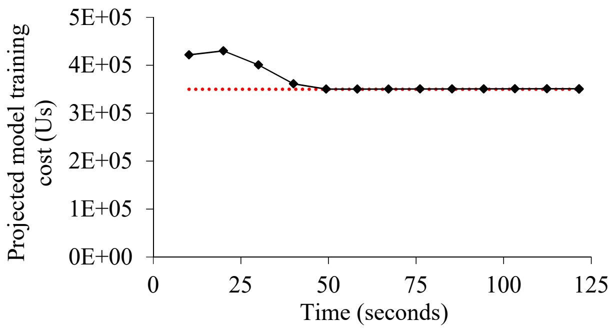


FIGURE 78. Model training cost projected over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and AdaGrad).

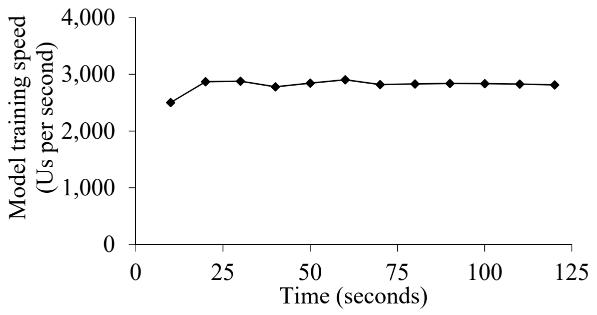


FIGURE 79. Model training speed over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and AdaGrad).

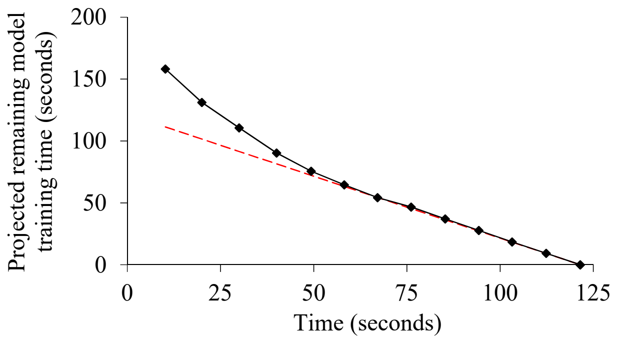


FIGURE 80. Remaining model training time projected over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and AdaGrad).

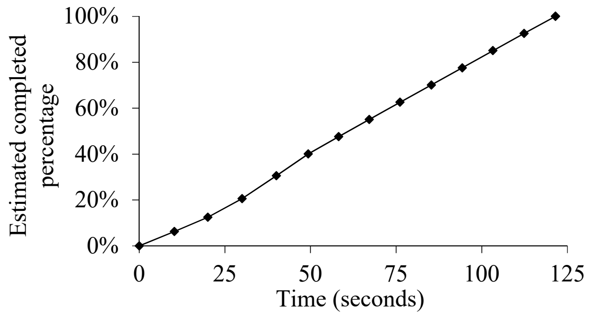


FIGURE 81. Completed percentage estimated over time (unloaded system test for training the GRU model using an exponential decay schedule for the learning rate and AdaGrad).

C. ADDITIONAL TEST RESULTS OF USING A STEP DECAY SCHEDULE FOR THE LEARNING RATE

1) UNLOADED SYSTEM TEST RESULTS FOR TRAINING GOOGLENET

In this section, we show the test results for model training using a step decay schedule for the learning rate. In each figure of this section, we use a vertical dash-dotted line to give the time when a learning rate decay occurred.

Using the RMSprop optimization method

In this section, we show the test results for training GoogLeNet on an unloaded system using a step decay schedule for the learning rate and the RMSprop optimization method. The performance results are shown in Fig. 82-85 and are similar to those shown in Fig. 31-34.

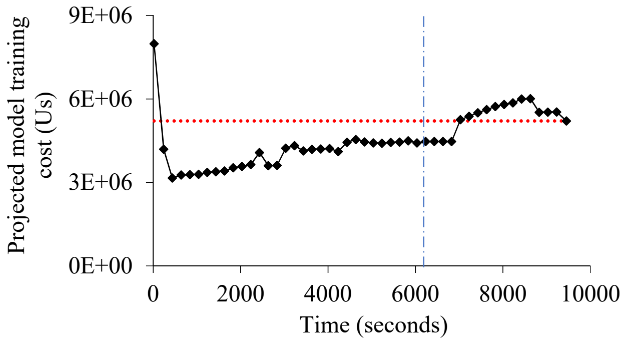


FIGURE 82. Model training cost projected over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and RMSprop).

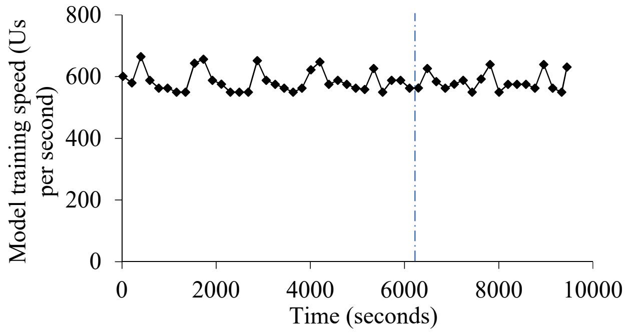


FIGURE 83. Model training speed over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and RMSprop).

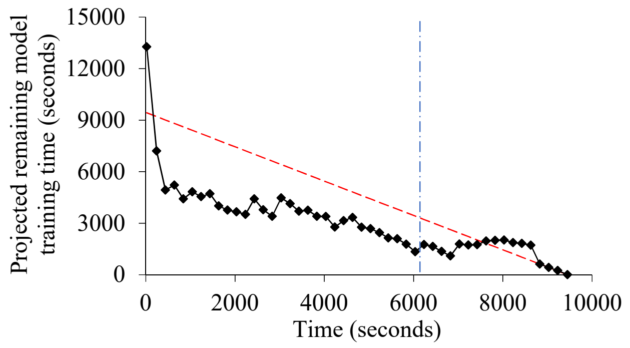


FIGURE 84. Remaining model training time projected over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and RMSprop).

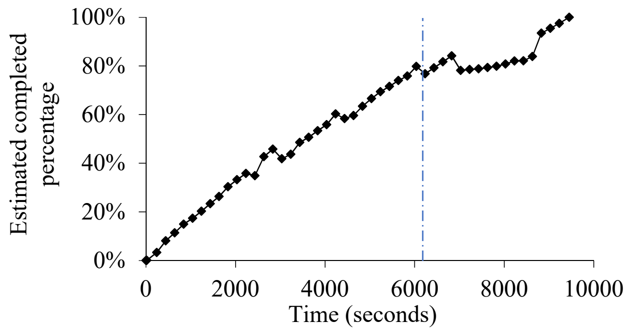


FIGURE 85. Completed percentage estimated over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and RMSprop).

Using the SGD optimization method

In this section, we show the test results for training GoogLeNet on an unloaded system using a step decay schedule for the learning rate and the SGD optimization method. The performance results are plotted in Fig. 86-89, showing the progress indicator made reasonably accurate projections.

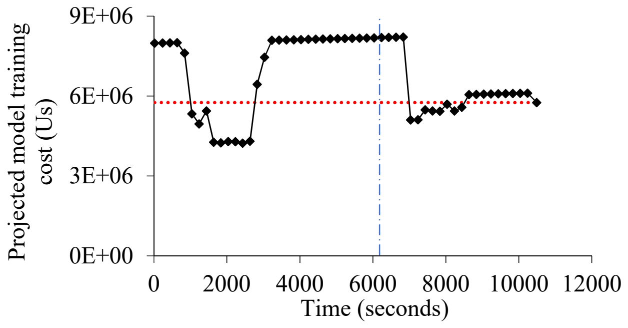


FIGURE 86. Model training cost projected over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and SGD).

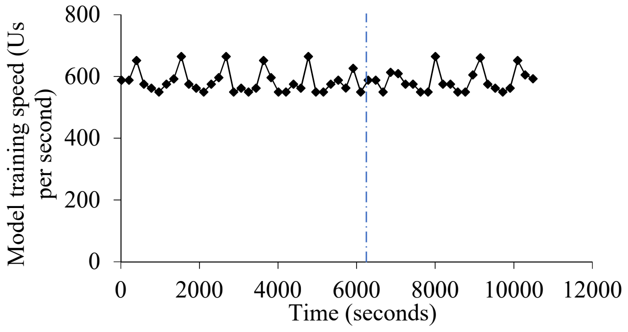


FIGURE 87. Model training speed over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and SGD).

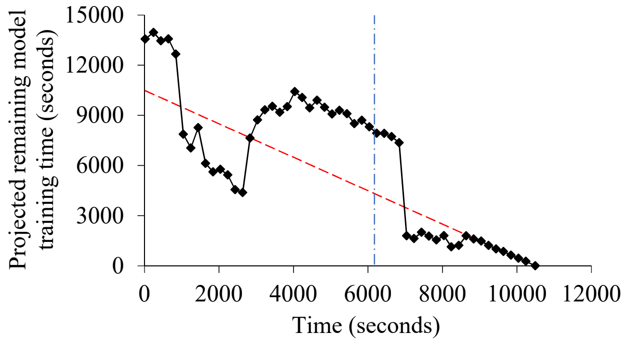


FIGURE 88. Remaining model training time projected over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and SGD).

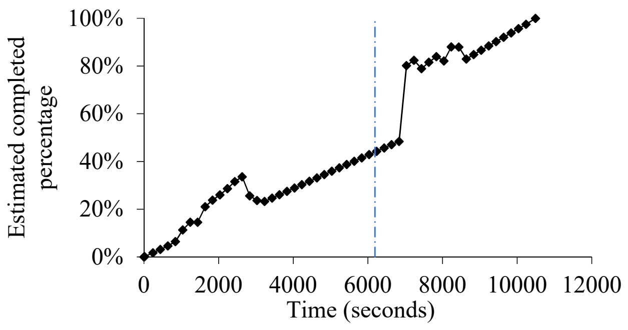


FIGURE 89. Completed percentage estimated over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and SGD).

Using the AdaGrad optimization method

In this section, we show the test results for training GoogLeNet on an unloaded system using a step decay schedule for the learning rate and the AdaGrad optimization method. The performance results are shown in Fig. 90-93 and are similar to those shown in Fig. 86-89.

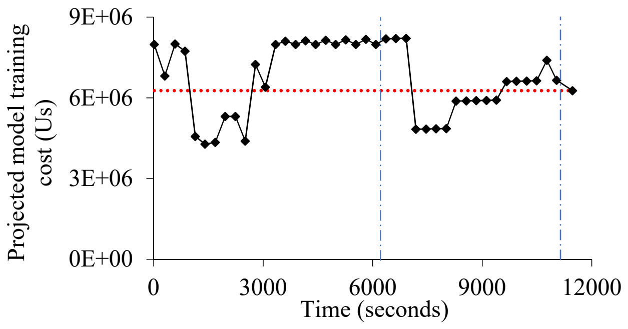


FIGURE 90. Model training cost projected over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and AdaGrad).

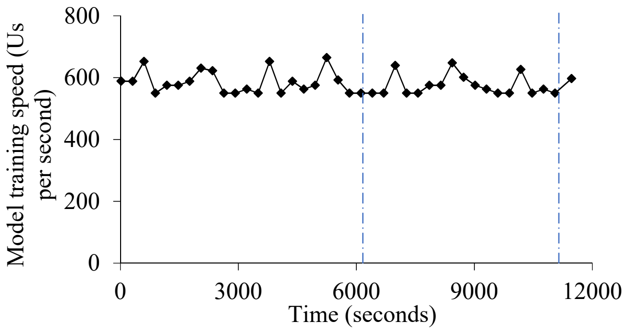


FIGURE 91. Model training speed over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and AdaGrad).

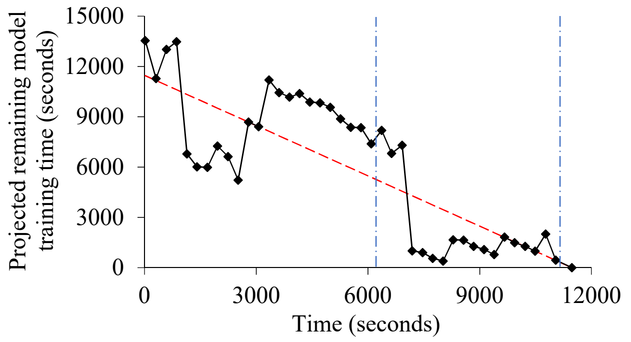


FIGURE 92. Remaining model training time projected over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and AdaGrad).

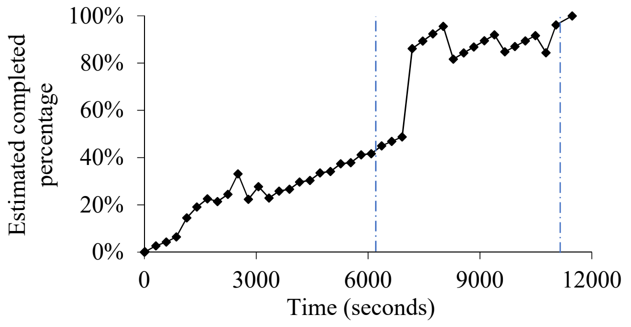


FIGURE 93. Completed percentage estimated over time (unloaded system test for training GoogLeNet using a step decay schedule for the learning rate and AdaGrad).

2) UNLOADED SYSTEM TEST RESULTS FOR TRAINING THE GRU MODEL

In each test shown in this section, early stopping occurred before the first decay point (see Fig. 5).

Using the Adam optimization method

In this section, we show the test results for training the GRU model on an unloaded system using a step decay schedule for the learning rate and the Adam optimization method. The performance results are plotted in Fig. 94-97, showing the progress indicator made reasonably accurate projections.

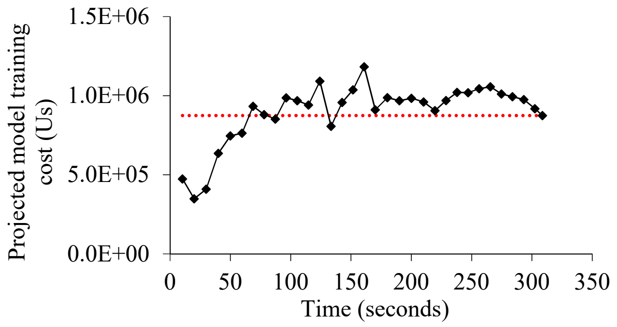


FIGURE 94. Model training cost projected over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and Adam).

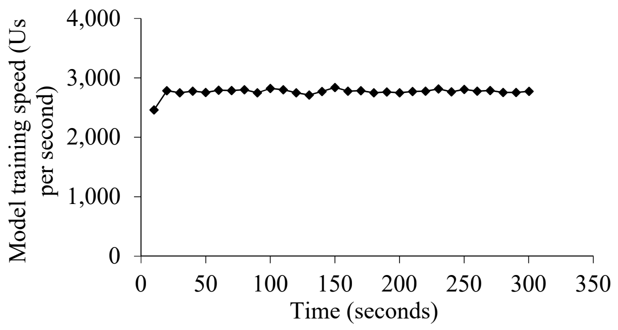


FIGURE 95. Model training speed over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and Adam).

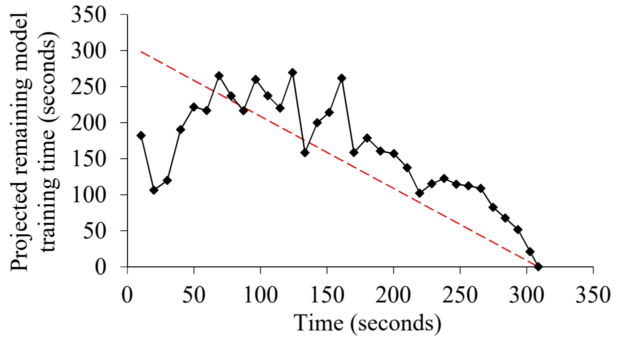


FIGURE 96. Remaining model training time projected over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and Adam).

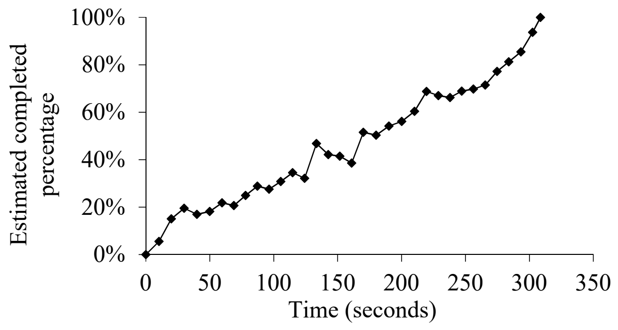


FIGURE 97. Completed percentage estimated over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and Adam).

Using the RMSprop optimization method

In this section, we show the test results for training the GRU model on an unloaded system using a step decay schedule for the learning rate and the RMSprop optimization method. The performance results are shown in Fig. 98-101 and are similar to those shown in Fig. 94-97.

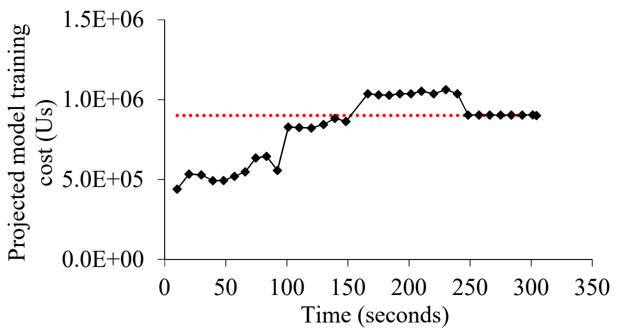


FIGURE 98. Model training cost projected over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and RMSprop).

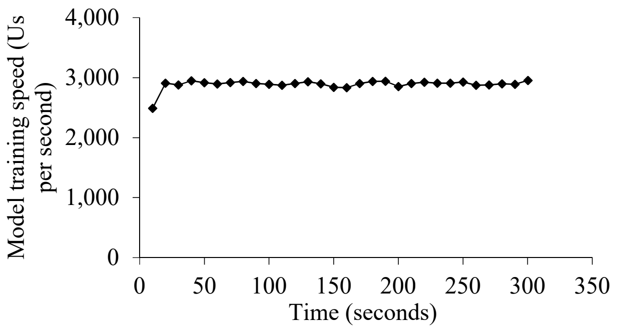


FIGURE 99. Model training speed over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and RMSprop).

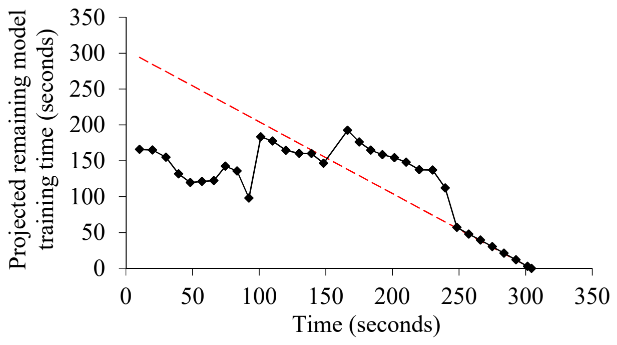


FIGURE 100. Remaining model training time projected over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and RMSprop).

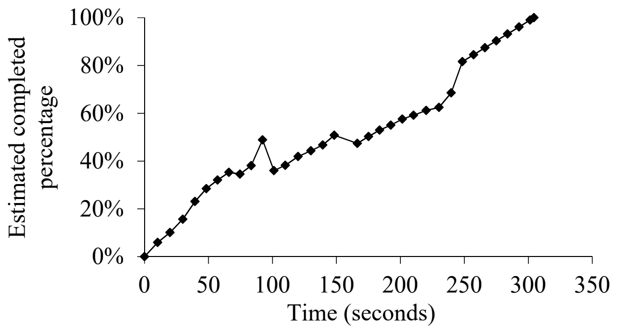


FIGURE 101. Completed percentage estimated over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and RMSprop).

Using the SGD optimization method

In this section, we show the test results for training the GRU model on an unloaded system using a step decay schedule for the learning rate and the SGD optimization method. The performance results are shown in Fig. 102-105. At 10 seconds, only three validation points were available, making it difficult to estimate the trend curve accurately. Hence, the progress indicator made inaccurate projections. After 20 seconds, the projections given by the progress indicator became much more accurate as more validation points became available.

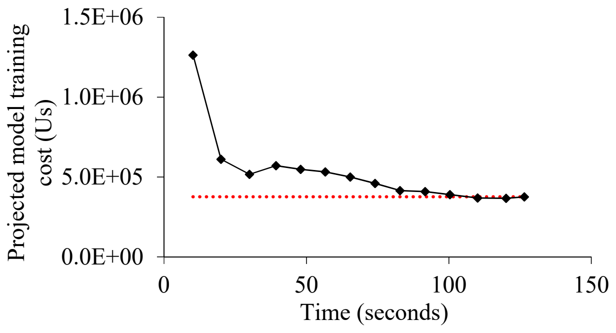


FIGURE 102. Model training cost projected over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and SGD).

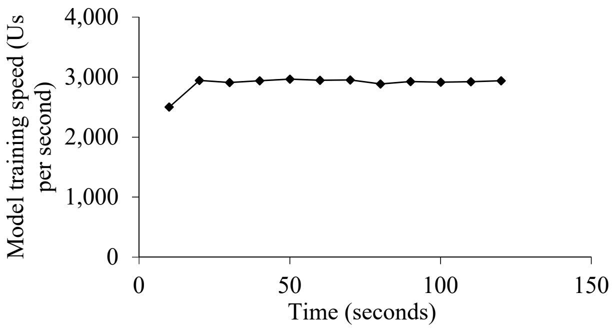


FIGURE 103. Model training speed over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and SGD).

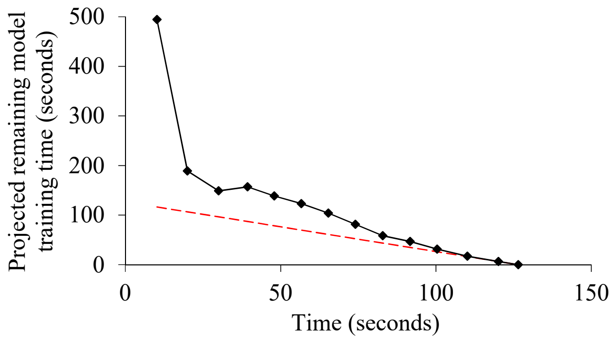


FIGURE 104. Remaining model training time projected over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and SGD).

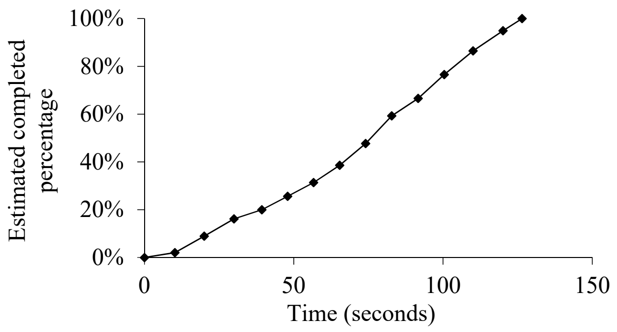


FIGURE 105. Completed percentage estimated over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and SGD).

Using the AdaGrad optimization method

In this section, we show the test results for training the GRU model on an unloaded system using a step decay schedule for the learning rate and the AdaGrad optimization method. The performance results are plotted in Fig. 106-109, showing the progress indicator made reasonably accurate projections.

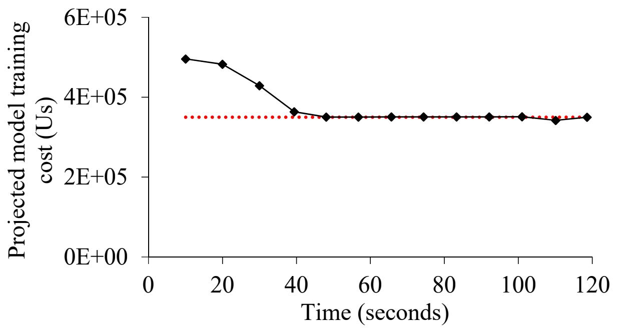


FIGURE 106. Model training cost projected over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and AdaGrad).

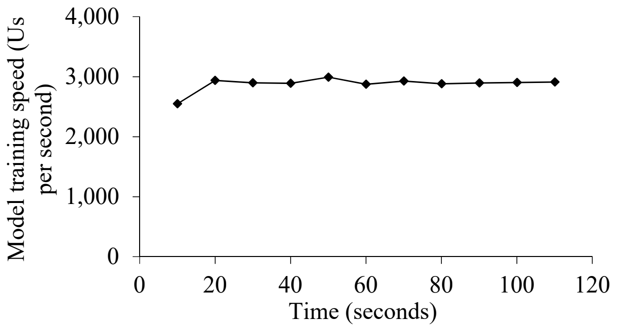


FIGURE 107. Model training speed over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and AdaGrad).

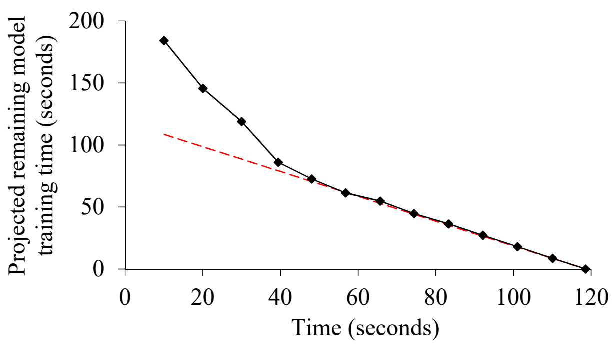


FIGURE 108. Remaining model training time projected over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and AdaGrad).

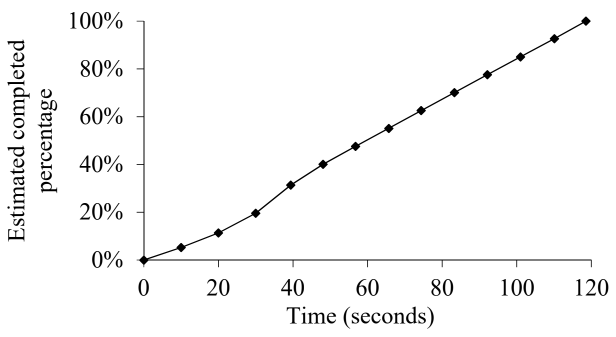


FIGURE 109. Completed percentage estimated over time (unloaded system test for training the GRU model using a step decay schedule on the learning rate and AdaGrad).

D. SUMMARY STATISTICS OF THE AVERAGE ESTIMATION ERROR ACROSS THE FIVE RUNS

TABLE IV
FOR EACH COMBINATION OF A DEEP LEARNING MODEL, A LEARNING RATE SCHEDULE, AND AN OPTIMIZATION METHOD IN THE UNLOADED SYSTEM TEST PRESENTED IN SECTIONS A TO C, THE SUMMARY STATISTICS OF THE AVERAGE ESTIMATION ERROR ACROSS THE FIVE RUNS

Deep learning model	Learning rate schedule	Optimization method	Average estimation error
GoogLeNet	fixed learning rate	SGD	0.093±0.012
	fixed learning rate	AdaGrad	0.093±0.008
	exponential decay	RMSprop	0.990±0.282
	exponential decay	SGD	0.897±0.274
	exponential decay	AdaGrad	0.632±0.197
	step decay	RMSprop	0.364±0.025
	step decay	SGD	0.540±0.160
	step decay	AdaGrad	0.552±0.196
GRU	fixed learning rate	RMSprop	0.275±0.101
	fixed learning rate	SGD	0.793±0.326
	fixed learning rate	AdaGrad	0.536±0.226
	exponential decay	RMSprop	0.284±0.046
	exponential decay	SGD	0.695±0.530
	exponential decay	AdaGrad	0.631±0.360
	step decay	Adam	0.333±0.050
	step decay	RMSprop	0.384±0.133
	step decay	SGD	0.568±0.217
	step decay	AdaGrad	0.304±0.169

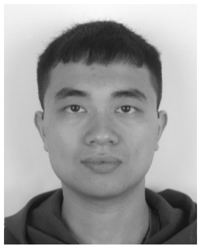
Recall that for each combination of a deep learning model, a learning rate schedule, and an optimization method in the unloaded system test, we trained the model five times, each in a separate run. For each combination presented in Sections A to C, we show the mean and the standard deviation of the average estimation error across the five runs in Table IV.

ACKNOWLEDGMENT
We thank Philip J. Brewster for helpful discussions.

AUTHORS' CONTRIBUTIONS
QD participated in designing the study and conducting literature review, wrote the paper's first draft, did the computer coding implementation, and performed experiments. GL conceptualized and designed the study, conducted literature review, and rewrote the whole paper. Both authors read and approved the final manuscript.

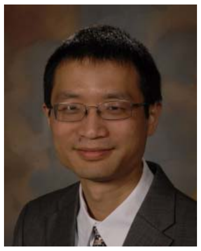
REFERENCES
[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.
[2] C. Sun, A. Shrivastava, S. Singh, and A. Gupta, "Revisiting unreasonable effectiveness of data in deep learning era," in *Proc. ICCV*, 2017, pp. 843-852.
[3] T. Weyand, I. Kostrikov, and J. Philbin, "Planet-photo geolocation with convolutional neural networks," in *Proc. ECCV*, 2016, pp. 37-55.
[4] T. Akiba, S. Suzuki, and K. Fukuda, "Extremely large minibatch SGD: training ResNet-50 on ImageNet in 15 minutes," in *Proc. NIPS Workshop on Deep Learning at Supercomputer Scale*, 2017.
[5] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and F.-F. Li, "ImageNet: a large-scale hierarchical image database," in *Proc. CVPR*, 2009, pp. 248-255.
[6] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann, 1993.
[7] G. Luo. "Toward a progress indicator for machine learning model building and data mining algorithm execution: a position paper," *SIGKDD Explorations*, vol. 19, no. 2, pp. 13-24, Dec. 2017.
[8] G. Luo, J. F. Naughton, and P. S. Yu, "Multi-query SQL progress indicators," in *Proc. EDBT*, 2006, pp. 921-941.
[9] "Keras integration with TQDM progress bars." GitHub. https://github.com/bstriner/keras-tqdm (accessed Feb. 10, 2020).
[10] "TensorBoard: visualization learning." GitHub. https://www.tensorflow.org/tensorboard/r1/summaries (accessed Feb. 10, 2020).
[11] G. Luo, "Progress indication for machine learning model building: a feasibility demonstration," *SIGKDD Explorations*, vol. 20, no. 2, pp. 1-12, Dec. 2018.
[12] S. Lau. "Learning rate schedules and adaptive learning rate methods for deep learning." Towards data science. https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1 (accessed Feb. 10, 2020).
[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: a system for large-scale machine learning," in *Proc. OSDI*, 2016, pp. 265-283.
[14] S. Chaudhuri, V. Narasayya, and R. Ramamurthy, "Estimating progress of execution for SQL queries," in *Proc. SIGMOD*, 2004, pp. 803-814.
[15] K. Lee, A. C. König, V. R. Narasayya, B. Ding, S. Chaudhuri, B. Ellwein, A. Eksarevskiy, M. Kohli, J. Wyant, P. Prakash, R. V. Nehme, J. Li, and J. F. Naughton, "Operator and query progress estimation in Microsoft SQL Server Live Query Statistics," in *Proc. SIGMOD*, 2016, pp. 1753-1764.
[16] G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke, "Increasing the accuracy and coverage of SQL progress indicators," in *Proc. ICDE*, 2005, pp. 853-864.
[17] G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke, "Toward a progress indicator for database queries," in *Proc. SIGMOD*, 2004, pp. 791-802.
[18] W. Lee, H. Oh, and K. Yi, "A progress bar for static analyzers," in *Proc. SAS*, 2014, pp. 184-200.
[19] G. Luo, T. Chen, and H. Yu, "Toward a progress indicator for program compilation," *Softw.: Pract. and Experience*, vol. 37, no. 9, pp. 909-933, July 2007.
[20] X. Xie, Z. Fan, B. Choi, P. Yi, S. S. Bhowmick, and S. Zhou, "PIGEON: progress indicator for subgraph queries," in *Proc. ICDE*, 2015, pp. 1492-1495.
[21] K. Morton, M. Balazinska, and D. Grossman, "ParaTimer: a progress indicator for MapReduce DAGs," in *Proc. SIGMOD*, 2010, pp. 507-518.
[22] K. Morton, A. L. Friesen, M. Balazinska, and D. Grossman, "Estimating the progress of MapReduce pipelines," in *Proc. ICDE*, 2010, pp. 681-684.
[23] G. Luo, "PredicT-ML: a tool for automating machine learning model building with big clinical data," *Health Inf. Sci. Syst.*, vol. 4, Article 5, Dec. 2016.
[24] G. Luo, B. L. Stone, M. D. Johnson, P. Tarczy-Hornoch, A. B. Wilcox, S. D. Mooney, X. Sheng, P. J. Haug, and F. L. Nkoy, "Automating construction of machine learning models with clinical big data: proposal rationale and methods," *JMIR Res. Protoc.*, vol. 6, no. 8, pp. e175, Aug. 2017.
[25] D. Justus, J. Brennan, S. Bonner, and A. S. McGough, "Predicting the computational cost of deep learning models," in *Proc. BigData*, 2018, pp. 3873-3882.
[26] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," in *Proc. NIPS*, 2012, pp. 2960-2968.
[27] T. Doan and J. Kalita, "Predicting run time of classification algorithms using meta-learning," *Int. J. Mach. Learn. and Cybern.*, vol. 8, no. 6, pp. 1929-1943, Dec. 2017.
[28] M. Reif, F. Shafait, and A. Dengel, "Prediction of classifier training time including parameter optimization," in *Proc. KI*, 2011, pp. 260-271.
[29] C. Yang, Y. Akimoto, D. W. Kim, and M. Udell, "OBOE: collaborative filtering for AutoML model selection," in *Proc. KDD*, 2019, pp. 1173-1183.
[30] M. Anthony and P. L. Bartlett, *Neural Network Learning: Theoretical Foundations*. New York, NY, USA: Cambridge Univ. Press, 2002.
[31] K. Fredenslund. "Computational complexity of neural networks." kasperfred.com. https://kasperfred.com/series/computational-complexity/computational-complexity-of-neural-networks (accessed Feb. 10, 2020).
[32] R. Livni, S. Shalev-Shwartz, and O. Shamir, "On the computational efficiency of training neural networks," in *Proc. NIPS*, 2014, pp. 855-863.
[33] L. Prechelt, "Early stopping-but when?" in *Neural Networks: Tricks of the Trade*. Berlin, Germany: Springer, 1996, pp. 55-69.
[34] D. Duvenaud, D. Maclaurin, and R. P. Adams, "Early stopping as nonparametric variational inference," in *Proc. AISTATS*, 2016, pp. 1070-1077.
[35] M. Mahsereci, L. Balles, C. Lassner, and P. Hennig, "Early stopping without a validation set," 2017, *arXiv: 1703.09580.*
[36] D.A. Berque and M. K. Goldberg, "Monitoring an algorithm's execution," in *Proc. DIMACS Workshop on Comput. Support for Discrete Math.,* USA, 1992, pp. 153-163.
[37] "EarlyStopping." TensorFlow. https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/keras/callbacks/EarlyStopping (accessed Feb. 10, 2020).
[38] R. L. Figueroa, Q. Zeng-Treitler, S. Kandula, and L. H. Ngo, "Predicting sample size required for classification performance," *BMC Med. Inform. Decis. Mak.*, vol. 12, Article 8, Feb. 2012.
[39] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed. New York, NY, USA: Springer, 2006.
[40] P. J. Huber and E. M. Ronchetti, *Robust Statistics*, 2nd ed. Hoboken, NJ, USA: Wiley, 2011.
[41] "SciPy.optimize.minimize." SciPy.org.

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize (accessed Mar. 20, 2020).
[42] "Optimize-minimize-TNC." SciPy.org. https://docs.scipy.org/doc/scipy/reference/optimize.minimize-tnc.html#optimize-minimize-tnc (accessed Mar. 20, 2020).
[43] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates." Ann. Math. Statist., vol. 29, no. 2, pp. 610-611, 1958.
[44] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. CVPR*, 2015, pp. 1-9.
[45] S. Purushotham, C. Meng, Z. Che, and Y. Liu, "Benchmarking deep learning models on large healthcare datasets," *J. Biomed. Inform.*, vol. 83, pp. 112-134, July 2018.
[46] "GoogLeNet-Inception." GitHub. https://github.com/conan7882/GoogLeNet-Inception (accessed Feb. 10, 2020).
[47] "Benchmarking_DL_MIMICIII." GitHub. https://github.com/USC-Melady/Benchmarking_DL_MIMICIII (accessed Feb. 10, 2020).
[48] D. P. Kingma and J. Ba, "Adam: a method for stochastic optimization." in *Proc. ICLR*, 2015.
[49] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proc. COMPSTAT*, 2010, pp. 177-186.
[50] S. Ruder, "An overview of gradient descent optimization algorithms," 2016, *arXiv:1609.04747*.
[51] J. C. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *JMLR*, vol. 12, pp. 2121-2159, Feb. 2011.
[52] A. Krizhevsky, "Learning multiple layers of features from tiny images," M.S. thesis, Dept. Comput. Sci., Univ. of Toronto, Toronto, Canada, 2009.
[53] A. E. W. Johnson, T. J. Pollard, L. Shen, L. H. Lehman, M. Feng, M. Ghassemi, B. Moody, P. Szolovits, L. A. Celi, and R. G. Mark, "MIMIC-III, a freely accessible critical care database," *Scientific Data*, vol. 3, Article 160035, May 2016.
[54] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. ICLR*, 2015.
[55] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735-1780, Nov. 1997.
[56] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and F.-F. Li, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211-252, Dec. 2015.
[57] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proc. ACL*, 2011, pp. 142-150.
[58] M. Sun, J. Li, Z. Guo, Y. Zhao, Y. Zheng, X. Si, and Z. Liu. "THUCTC: an efficient Chinese text classifier." http://thuctc.thunlp.org (accessed Mar. 27, 2020).
[59] "Text-classification-cnn-rnn." GitHub. https://github.com/gaussic/text-classification-cnn-rnn (accessed Mar. 27, 2020).
[60] "VGG-cifar," GitHub. https://github.com/conan7882/VGG-cifar (accessed Mar. 8, 2020).
[61] "LSTM-Sentiment-Analysis," GitHub. https://github.com/adeshpande3/LSTM-Sentiment-Analysis (accessed Mar. 8, 2020).
[62] S. Chaudhuri, R. Kaushik, and R. Ramamurthy, "When can we trust progress estimators for SQL queries?" in *Proc. SIGMOD*, 2005, pp. 575-586.
[63] A. Efrat, Q. Fan, and S. Venkatasubramanian, "Curve matching, time warping, and light fields: new algorithms for computing similarity between curves," *J. Math. Imag. and Vision*, vol. 27, no. 3, pp. 203-216, Apr. 2007.
[64] F. Hu and J. V. Zidek, "The weighted likelihood," *Can. J. Statist.*, vol. 30, no. 3, pp. 347-371, Sep. 2002.



QIFEI DONG received the B.S degree in electrical engineering from Zhejiang University, Hangzhou, Zhejiang Province, P.R. China, in 2016 and the M.S degree in electrical and computer engineering from the University of Michigan, Ann Arbor, MI, USA, in 2018. He is currently pursuing the Ph.D. degree in biomedical informatics and medical education at the University of Washington, Seattle, WA, USA.

Since 2018, he has been a Research Assistant with the University of Washington Clinical Learning, Evidence and Research Center for Musculoskeletal Disorders. His research interests include machine learning, computer vision, natural language processing, and clinical informatics.



GANG LUO received the B.S. degree in computer science from Shanghai Jiaotong University, Shanghai, P.R. China, in 1998, and the PhD degree in computer science from the University of Wisconsin-Madison, Madison, WI, USA, in 2004.

From 2004 to 2012, he was a Research Staff Member at IBM T.J. Watson Research Center, Hawthorne, NY, USA. From 2012 to 2016, he was an Assistant Professor in the Department of Biomedical Informatics at the University of Utah, Salt Lake City, UT, USA. He is currently an Associate Professor in the Department of Biomedical Informatics and Medical Education at the University of Washington, Seattle, WA, USA. He is the author of over 70 papers. His research interests include machine learning, information retrieval, database systems, and health informatics.