ONLINE BATCH SELECTION FOR FASTER TRAINING OF NEURAL NETWORKS

Ilya Loshchilov & Frank Hutter
Univesity of Freiburg
Freiburg, Germany,
{ilya,fh}@cs.uni-freiburg.de

ABSTRACT

Deep neural networks are commonly trained using stochastic non-convex optimization procedures, which are driven by gradient information estimated on fractions (batches) of the dataset. While it is commonly accepted that batch size is an important parameter for offline tuning, the benefits of online selection of batches remain poorly understood. We investigate online batch selection strategies for two state-of-the-art methods of stochastic gradient-based optimization, AdaDelta and Adam. As the loss function to be minimized for the whole dataset is an aggregation of loss functions of individual datapoints, intuitively, datapoints with the greatest loss should be considered (selected in a batch) more frequently. However, the limitations of this intuition and the proper control of the selection pressure over time are open questions. We propose a simple strategy where all datapoints are ranked w.r.t. their latest known loss value and the probability to be selected decays exponentially as a function of rank. Our experimental results on the MNIST dataset suggest that selecting batches speeds up both AdaDelta and Adam by a factor of about 5.

1 INTRODUCTION

Deep neural networks (DNNs) are currently the best-performing method for many classification problems, such as object recognition from images (Krizhevsky et al., 2012; Donahue et al., 2014) or speech recognition from audio data (Deng et al., 2013). The standard method to train DNNs is backpropagation via stochastic gradient descent (SGD) based on fixed-size mini-batches of the data. This training step is the computational bottleneck of DNNs: on large datasets (where DNNs perform particularly well), it requires several days, even on high-performance GPUs, and any speedups would be of substantial value to the booming field of DNNs.

In light of the computational burden of DNN training, we find it surprising that popular DNN optimization methods do not implement procedures to adaptively choose which datapoints to include in a batch. Instead, it is common in SGD to fix the batch size and iteratively sweep the algorithm over all datapoints. Here, we propose and evaluate online batch selection methods to help SGD focus on the most relevant training datapoints and progress faster. More specifically, we sample the datapoints for each batch from a non-uniform probability distribution that depends on the loss previously computed for each datapoint under consideration.

We first formally describe stochastic gradient descent and the current state-of-the-art DNN optimization procedures AdaDelta (Zeiler, 2012) and Adam (Kingma & Ba, 2014). Then, we describe our proposed online batch selection procedure and demonstrate that it can substantially speed up both AdaDelta and Adam.

2 STOCHASTIC GRADIENT DESCENT (SGD)

The training of a DNN with n free parameters can be formulated as the problem of minimizing a function $f : \mathbb{R}^n \to \mathbb{R}$. Following the notation of Hazan et al. (2015), we define a loss function $\psi : \mathbb{R}^n \to \mathbb{R}$ for each training datapoint; the distribution of training datapoints then induces a

---

Algorithm 1 The original AdaDelta algorithm

1: **given** $\rho = 0.95, \epsilon = 10^{-6}$
2: **initialize** time step $t \leftarrow 0$, parameter vector $\boldsymbol{x}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{s}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$
3: **repeat**
4:　　$t \leftarrow t + 1$
5:　　$\nabla f_t(\boldsymbol{x}_{t-1}) \leftarrow SelectBatch(\boldsymbol{x}_{t-1})$　　　▷ select batch and return the corresponding gradient
6:　　$\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{x}_{t-1})$
7:　　$\boldsymbol{v}_t \leftarrow \rho\boldsymbol{v}_{t-1} + (1-\rho)\boldsymbol{g}_t^2$　　　　　　　▷ here and below all operations are element-wise
8:　　$\Delta\boldsymbol{x}_t \leftarrow -\dfrac{\sqrt{\boldsymbol{s}_{t-1}+\epsilon}}{\sqrt{\boldsymbol{v}_t+\epsilon}}\boldsymbol{g}_t$
9:　　$\boldsymbol{s}_t \leftarrow \rho\boldsymbol{s}_{t-1} + (1-\rho)\Delta\boldsymbol{x}_t^2$
10:　　$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} + \Delta\boldsymbol{x}_t$
11: **until** *stopping criterion is met*
12: **return** optimized parameters $\boldsymbol{x}_t$

---

distribution over functions $\mathcal{D}$, and the overall function f we aim to optimize is the expectation of this distribution:

$$f(\boldsymbol{x}) := \mathbb{E}_{\psi \sim \mathcal{D}}\left[\psi(\boldsymbol{x})\right]. \tag{1}$$

The commonly used procedure to optimize f is to iteratively adjust $\boldsymbol{x}_t$ (the parameter vector at time step t) using gradient information obtained on a relatively small batch of size b. More specifically, at each time step t and for a given $\boldsymbol{x}_t \in \mathbb{R}^n$, a batch $\left\{\psi_{i=1}^b\right\} \sim \mathcal{D}^b$ is selected to compute $\nabla f_t(\boldsymbol{x}_t)$, where $f_t(\boldsymbol{x}_t) = \frac{1}{b}\sum_{i=1}^b \psi_i(\boldsymbol{x}_t)$. The Stochastic Gradient Descent (SGD) procedure then becomes a natural extension of the Gradient Descent (GD) to stochastic optimization of f as follows:

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \eta_t \nabla f_t(\boldsymbol{x}_t), \tag{2}$$

where η_t is a learning rate. One would like to use second-order information

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \eta_t \boldsymbol{H}_t^{-1} \nabla f_t(\boldsymbol{x}_t), \tag{3}$$

but this is often infeasible since the computation and storage of the Hessian $\boldsymbol{H}_t$ is intractable for large n. The usual way to deal with this problem by using limited-memory quasi-Newton methods such as L-BFGS (Liu & Nocedal, 1989) is not currently in favor in deep learning, not the least due to (i) the stochasticity of $\nabla f_t(\boldsymbol{x}_t)$, (ii) ill-conditioning of f and (iii) the presence of saddle points as a result of the hierarchical geometric structure of the parameter space (Fukumizu & Amari, 2000). Despite some recent progress in understanding and addressing the latter problems (Bordes et al., 2009; Dauphin et al., 2014; Choromanska et al., 2014; Dauphin et al., 2015), state-of-the-art optimization techniques tend to approximate the inverse Hessian in a reduced way, e.g., by considering only its diagonal to achieve adaptive learning rates.

In this paper, we argue that not only the update of $\boldsymbol{x}_t$ given $\nabla f_t(\boldsymbol{x}_t)$ is crucial, but the selection of the batch $\left\{\psi_{i=1}^b\right\} \sim \mathcal{D}^b$ used to compute $\nabla f_t(\boldsymbol{x}_t)$ also greatly contributes to the overall performance.

3　ADADELTA AND ADAM

We start by describing the Adadelta and Adam algorithms, which we both build upon and use as the baseline for our experiments. Other related work (most of which appeared as parallel ICLR submissions) is described in Section 7.

The AdaDelta algorithm (Zeiler, 2012) is based on AdaGrad (Duchi et al., 2011), where the learning rate η_t is set to be $\eta/\sqrt{\sum_{i=1}^t (\nabla f_i(\boldsymbol{x}_i))^2}$ with the purpose to estimate the diagonal of the Hessian of

---

Algorithm 2 The original Adam algorithm

1: **given** $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$
2: **initialize** time step $t \leftarrow 0$, parameter vector $\boldsymbol{x}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$
3: **repeat**
4:   $t \leftarrow t + 1$
5:   $\nabla f_t(\boldsymbol{x}_{t-1}) \leftarrow SelectBatch(\boldsymbol{x}_{t-1})$ $\triangleright$ select batch and return the corresponding gradient
6:   $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{x}_{t-1})$
7:   $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$ $\triangleright$ here and below all operations are element-wise
8:   $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
9:   $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t/(1 - \beta_1^t)$ $\triangleright$ here, β_1 is taken to the power of t
10:   $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t/(1 - \beta_2^t)$ $\triangleright$ here, β_2 is taken to the power of t
11:   $\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \alpha \hat{\boldsymbol{m}}_t/(\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon)$
12: **until** *stopping criterion is met*
13: **return** optimized parameters $\boldsymbol{x}_t$

---

the functions f_i (Duchi et al., 2011). This idea can be traced back to variable metric methods, such as the BFGS family of algorithms (Fletcher, 1970). The accumulation of gradient amplitudes from $t = 1$ can negatively impact the search if the old information is irrelevant for the current objective function landscape. Similarly to the approach proposed by Schaul et al. (2012), AdaDelta attempts to overcome this issue by decaying the information about gradient amplitudes (stored in vector $\boldsymbol{v}_t$, see line 7 in Algorithm 1) with a factor of $1 - \rho$ defining the time window. Additionally, AdaDelta stores the information about parameter changes $\Delta \boldsymbol{x}_t$ (stored in vector $\boldsymbol{s}_t$, see line 9) to recompute the new $\Delta \boldsymbol{x}_t$ as $-\frac{\sqrt{\boldsymbol{s}_{t-1}+\epsilon}}{\sqrt{\boldsymbol{v}_t+\epsilon}}\boldsymbol{g}_t$. The purpose of this normalization is to capture the change of gradient amplitudes and to adjust the learning rates accordingly.

The Adam algorithm (Kingma & Ba, 2014) considers two momentum vectors, the first momentum $\boldsymbol{m}_t$ for gradients with a decay factor $1 - \beta_1$ (see line 7 in Algorithm 2) and the second momentum $\boldsymbol{v}_t$ for gradient amplitudes with a decay factor $1 - \beta_2$ (see line 8). In order to overcome the initialization bias introduced by setting $\boldsymbol{m}_{t=0}$ and $\boldsymbol{v}_{t=0}$ to zero vectors, the authors proposed to use bias-corrected estimates $\hat{\boldsymbol{m}}_t$ and $\hat{\boldsymbol{v}}_t$ (see lines 9-10). The change of $\boldsymbol{x}_t$ is performed according to the first bias-corrected momentum $\hat{\boldsymbol{m}}_t$, with the learning rates set according to the second bias-corrected momentum $\hat{\boldsymbol{v}}_t$.

The principal reason for selecting AdaDelta and Adam for our experiments is their claimed robustness which should reduce the problem of hyperparameter tuning to obtain satisfactory performance.

4 ONLINE BATCH SELECTION

All approaches mentioned in the previous section attempt to deal with stochasticity arising from the fact that the noise-less $\nabla f(\boldsymbol{x}_t)$ is often intractable to compute. Instead, $\nabla f_t(\boldsymbol{x}_t)$ is computed w.r.t. a batch $\{\psi_{i=1}^b\} \sim \mathcal{D}^b$ (see line 5 in Algorithms 1 and 2). It is common to reshuffle the whole dataset once per epoch and divide it into a number of batches with size b and thus to schedule the appearance of datapoints. As different datapoints correspond to different loss function values (e.g., some are more difficult to classify), gradient amplitudes and directions may also appear to be significantly different. This motivated the use of momentum both for gradient directions and amplitudes to smooth the incoming information.

We propose to online select batches (both size and datapoints) for an algorithm $\mathcal{A}$ to maximize its progress (e.g., defined by the objective function f or cross-validation error) over the resource budget (e.g., the number of evaluated datapoints or time). More formally, this problem can be formulated in the framework of reinforcement learning (Sutton & Barto, 1998). In this paper, we provide an initial study of this problem in the context of deep learning to explore possible design choices starting from the simplest ones. Related works on batch selection are discussed in section 7.

Let us assume for simplicity that the number of available training datapoints is bounded by N. At each time step t, the probability of the i-th datapoint to be selected to compose a batch of size b

---

Algorithm 3 Batch Selection Procedure SelectBatch()

1: **given** parameter vector $\boldsymbol{x} \in \mathbb{R}^n$
2: **global parameters** number of datapoints N, datapoints d_i for $i = 1, \ldots, N$; batch size b; counters c, c_e, c_s, c_r; two-dimensional array l with $l^{(1)}_{(1:N)}$ storing loss values and $l^{(2)}_{(1:N)}$ storing datapoint indexes; selection pressure s_e; selection probabilities p_i and cumulative selection probabilities a_i defined for i-th loss ranked datapoints; period of sorting T_s; frequency of loss recomputation r_{freq} for $r_{ratio} \cdot N$ datapoints with the greatest known loss values
3: **first call initialization** $c \leftarrow 0, c_e \leftarrow -N, c_s \leftarrow 0, c_r \leftarrow 0, l^{(1)}_{(1:N)} = \infty, l^{(2)}_{(1:N)} = 1 : N$
4: $c \leftarrow c + b$
5: **if** $c - c_e > N$ **then**
6:    $c_e \leftarrow c, e \leftarrow c_e$ div N ▷ once per epoch, we recompute selection probabilities
7:    use Eq. (5) to compute s_e ▷ $O(1)$
8:    use Eq. (4) to compute $p_i, i = 1, \ldots, N$ ▷ $O(N)$
9:    $a_j \leftarrow \sum_{i=1}^{j} p_i$ for $j = 1, \ldots, N$ ▷ $O(N)$
10: **if** $c - c_s > T_s$ **then**
11:    $c_s \leftarrow c$ ▷ every T_s evaluated datapoints, we sort datapoints w.r.t. the latest known loss
12:    $l \leftarrow sort(l)$ in descending order w.r.t. loss values stored in array $l^{(1)}_{(:)}$ ▷ $O(N \log(N))$
13: **if** $c - c_r > N/r_{freq}$ **then**
14:    $c_r \leftarrow c$ ▷ r_{freq} times per epoch, we recompute loss values for the top $r_{ratio} \cdot N$ datapoints
15:    **for** i = 1 to $r_{ratio} \cdot N$ **do**
16:       $l^{(1)}_i \leftarrow \psi_i(\boldsymbol{x})$ w.r.t. datapoint with index $l^{(2)}_i$ ▷ we do it in biggest possible batches
17:    $l \leftarrow sort(l)$ in descending order w.r.t. loss values stored in array $l^{(1)}_{(:)}$ ▷ $O(N \log(N))$
18: **for** i = 1 to b **do**
19:    **if** Default random selection **then**
20:       sample datapoint index $d_i \in [1, N]$ uniformly at random
21:    **if** Default shuffle selection **then**
22:       sample datapoint index $d_i \in [1, N]$ uniformly at random without repetitions in one epoch
23:    **if** Loss-dependent selection **then**
24:       sample random $r \in [0, 1)$, find the lowest rank index i_{sel} s.t. $r < a_{i_{sel}}$ ▷ $O(\log(N))$
25:       $d_i \leftarrow l^{(2)}_{i_{sel}}$
26:    associate $\psi_i(\boldsymbol{x})$ with the loss function computed on the datapoint with index d_i
27: $f \leftarrow \frac{1}{b} \sum_{i=1}^{b} \psi_i(\boldsymbol{x})$ ▷ forward pass
28: **for** i = 1 to b **do**
29:    $l^{(1)}_{i_{sel}} \leftarrow \psi_i(\boldsymbol{x})$ ▷ latest known loss
30: **return** $\nabla f(\boldsymbol{x})$ ▷ compute and return gradient

---

is denoted by p_i. By setting p_i to 0 for all already selected datapoints and to $\frac{1}{N+1-t}$ otherwise, the whole set will be enumerated in N/b steps (denoted here as one epoch). Clearly, if we set the selection probability p_i of some datapoints to 0, then, we reduce the size of the training set by eliminating these datapoints from the batch selection. A smaller training set is typically easier to fit but also to over-fit. A possible strategy is to first select the datapoints with the greatest contributions to the objective function; e.g., one could always select the datapoint with the greatest loss. However, after the exploitation of that datapoint is performed and its loss is (likely) reduced, the datapoint with the next greatest loss would need to be identified and any datapoint from the training set is a potential candidate. While one can rely on the assumption that ranks of losses over datapoints change slowly in time, the exploitation-exploration dilemma is inevitable.

The procedure we propose for batch selection is given in Algorithm 3, and we can integrate it into AdaDelta and Adam by calling it in line 5 of Algorithms 1 and 2. When datapoints are shuffled or selected uniformly at random (i.e., when the default AdaDelta and Adam are used), the first 17 lines of Algorithm 3 should be skipped.

We propose a simple rank-based batch selection strategy, where all training datapoints k are ranked (sorted) in descending order w.r.t. their latest computed $\psi_k(\boldsymbol{x})$ (see lines 12 and 17 in Algorithm 3). Each i-th ranked datapoint is selected with probability p_i. We propose to exponentially decay

p_i with its rank i such that $p_N = \frac{p_1}{s_e}$, where s_e is the ratio between the greatest and smallest probabilities of selection at epoch e. Thus, between any i and $i+1$, the probability drops by a factor of $\exp(\log(s_e)/N)$. When normalized to sum up to 1.0, the probability of i-th ranked datapoint to be selected is defined as

$$p_i = \frac{1/\exp(\log(s_e)/N)^i}{\sum_{i=1}^{N} 1/\exp(\log(s_e)/N)^i}, \tag{4}$$

Intuitively, this selection pressure parameter s_e affects the effective size of the training data we use. Let N_{eff} denote the number of unique datapoints selected among the last N selections; then, N_{eff} is likely (but not necessarily, see section 11.1 in the supplementary material) to decrease when s increases.

To imitate the increase of N_{eff} at each epoch e, from epoch e_0 to epoch e_{end}, we exponentially decrease the selection pressure from s_{e_0} to $s_{e_{end}}$ with

$$s_e = s_{e_0} \exp(\log(s_{e_{end}}/s_{e_0})/(e_{end} - e_0))^{e_{end}-e_0}, \tag{5}$$

and then substitute it into (4) to compute it once per epoch. The realization of datapoint selection is quite straightforward: given an array $a_j \leftarrow \sum_{i=1}^{j} p_i$ for $j = 1, \ldots, N$ (once per epoch when s_e is updated, see lines 7-9), we uniformly at random generate $r \in [0, 1)$ and find the lowest index i_{sel} such that $r < a_{i_{sel}}$ (see line 24). The datapoint to be selected is the one whose rank w.r.t. the latest known loss value is i_{sel} (see line 25).

The latest loss computed for a datapoint is only an estimate of its current loss. In order to improve the quality of the ranking, we propose to periodically, r_{freq} times per epoch, recompute the loss for $r_{ratio} \cdot N$ datapoints with the greatest latest known loss (see lines 16-17).

5 TIME COMPLEXITY

The computational complexity of datapoint selection scales as $O(\log(N))$ per datapoint using a bisection-based search in the array a whose values are increasing with rank. The sorting procedure scales as $O(N \log(N))$, but is performed periodically every T_s evaluated datapoints (which in turn reduces the constant factor because the array is partially sorted). The time overhead of the selection procedure without recomputing loss values (see lines 16-17) is roughly 5% when measured on a GPU for MNIST. If the cost of computing training and validation losses (for plotting) is included, then the relative cost is smaller than 5%.

The procedure of recomputing loss values (see lines 16-17) is optional. It is parameterized by its frequency r_{freq} and the amount of datapoints $r_{ratio} \cdot N$; it costs $r_{freq} \cdot r_{ratio}$ forward passes per datapoint. In practice, if the maximum feasible batch size is used and the recomputation is performed every two epochs ($r_{freq} = 0.5$) for the whole dataset ($r_{ratio} = 1.0$), then the total overhead is expected to be roughly 20% (i.e., by a factor of 1.2 more expensive) w.r.t. the default algorithms. If the cost of computing training and validation losses (for plotting) is included (e.g., it is excluded in Figure 1), then the relative cost is smaller than 20%.

6 EXPERIMENTAL RESULTS

We empirically investigate the impact of the proposed online batch selection strategies when combined with AdaDelta and Adam. We conducted our experiments on the standard MNIST dataset of handwritten digits (LeCun et al., 1998) with 50.000 training, 10.000 validation and 10.000 test datapoints. We do not employ any data augmentation and pre-processing procedures. All experiments were performed using the python library Lasagne[1]; we provide the source code to reproduce our experiments online at https://sites.google.com/site/batchsel/ (anonymous for the
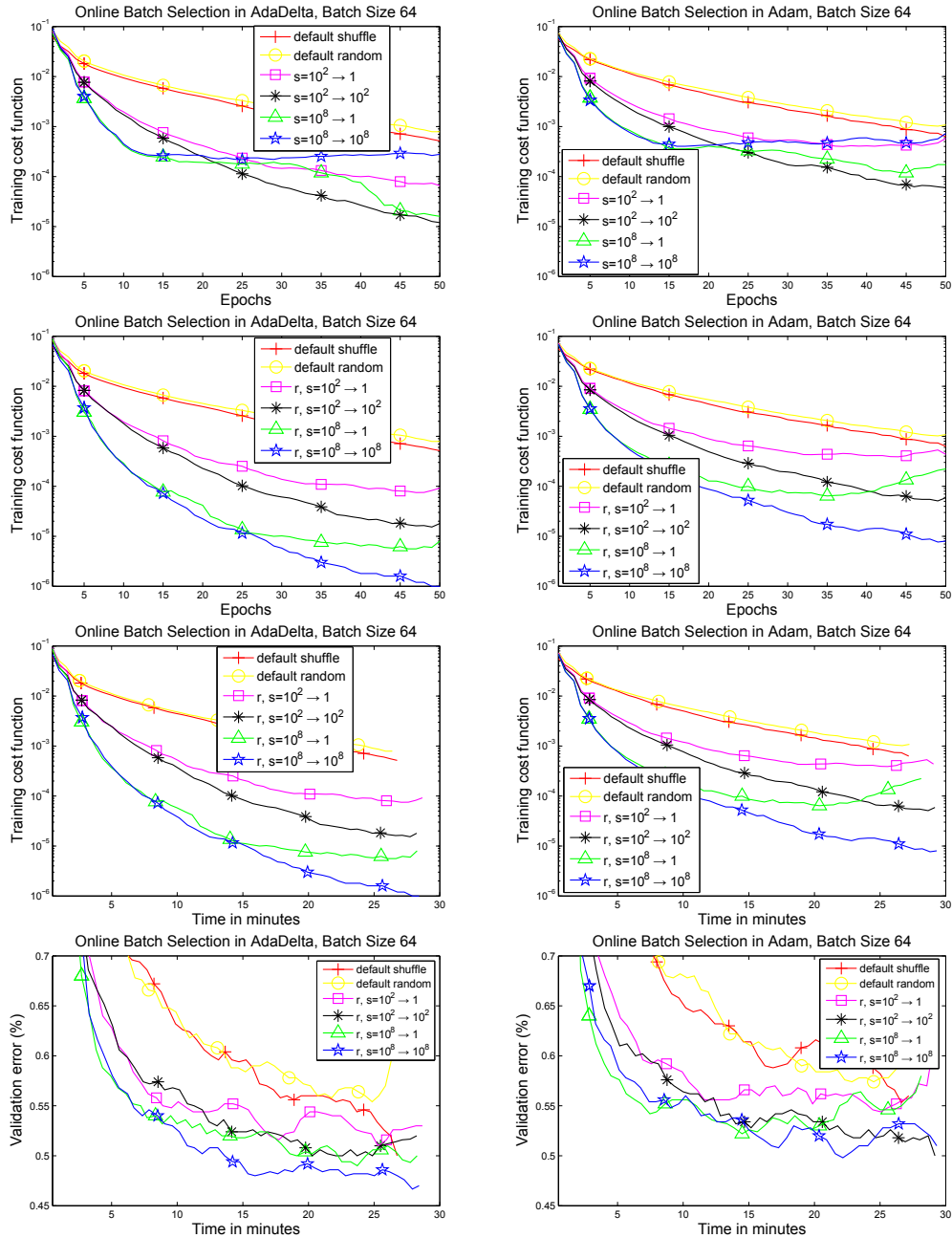
---

[1]https://github.com/Lasagne/Lasagne

Figure 1: Convergence curves of AdaDelta (**Left Column**) and Adam (**Right Column**) on MNIST dataset. The original algorithms are denoted by "default shuffle" (respectively, "default with random") when datapoints are shuffled and selected sequentially (respectively, uniformly at random). The value of s denotes the ratio of probabilities to select the training datapoint with the greatest latest known loss rather than the smallest latest known loss. Legends with $s = s_{e_0} \rightarrow s_{e_{end}}$ correspond to an exponential change of s from s_{e_0} to $s_{e_{end}}$ as a function of epoch index e, see Eq. (5). Legends with the prefix "r" correspond to the case when the loss values for $r_{ratio} \cdot N = 1.0N$ datapoints with the greatest latest known loss are recomputed $r_{freq} = 0.5$ times per epoch. All curves are computed on the whole training set and smoothed by a moving average of the median of 11 runs.
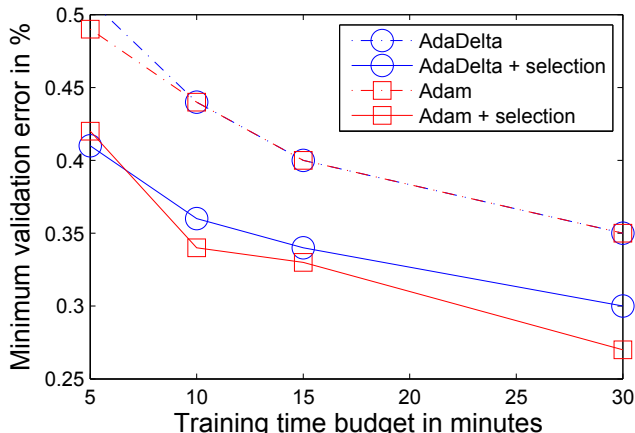
Figure 2: Minimum validation error found on MNIST for deep neural network trained with a given budget of time. The results correspond to the best hyperparameter settings found by CMA-ES after 1000 function evaluations. The original Adam and Adadelta algorithms are denoted by dotted lines, their versions with batch selection are denoted by solid lines.

reviewers). In all experiments, we use Batch Normalization (Ioffe & Szegedy, 2015) as implemented by Courbariaux et al. (2015) in Lasagne because i) we found that it consistently improves the error rate by roughly 0.1-0.15%, and ii) to demonstrate the compatibility of our approach with the existing acceleration techniques.

We performed our experiments with various values of the batch size b, but mainly report the results for $b = 64$ (see Figure 1) because they demonstrate better generalization errors over time. The results for $b = 16$ and $b = 256$ are given in the supplementary material (Figures 5 and 6, respectively).

Figure 1 shows that a 5-fold speedup can be achieved i) during the first 10 epochs if the recomputation procedure is not considered (see our early experiments in Figure 1 (top row)), and, ii) during the whole run if the recomputation procedure is considered (see the remaining rows in Figure 1). Figure 1 (top row) is reported mainly to support the motivation for the recomputation procedure.

Figure 1 (top row) shows the cost function (cross-entropy loss) over epochs for the original AdaDelta and Adam (when datapoints are shuffled and selected sequentially or uniformly at random) and their accelerated versions with our batch selection procedure. Our original approach was to have a fixed s (see, e.g., $s = 10^2 \rightarrow 10^2$ and $s = 10^8 \rightarrow 10^8$ in the figure) and we found that $s = 10^2$ significantly speeds up both AdaDelta and Adam. Greater values, such as $s = 10^8$, lead to even faster convergence during the first 10 epochs. However, the cost function stops decaying at later epochs and a divergence may occur (more likely for greater values of s). This effect might partially explain why previous attempts (if any) to perform batch selection were not successful: the use of very strong selection might lead to divergence. To reduce this problem, we introduced a decreasing s over epochs as described in Eq. (5). The exponential decrease of s from 10^2 to 1 over 50 epochs (denoted as $s = 10^2 \rightarrow 1$) corresponds to a standard uniform selection at the last epoch such that most of the training datapoints will be considered. Using $s = 10^2 \rightarrow 1$ corresponds to a weaker selection and a reduced speedup compared to $s = 10^2 \rightarrow 10^2$. However, $s = 10^8 \rightarrow 1$ combines the fast convergence at the first 10 epochs with a steady improvement at the later epochs.

To further improve the convergence for greater values of s, we proposed to additionally, $r_{freq} = 0.5$ times per epoch, recompute the loss for $r_{ratio} \cdot N = 1.0N$ datapoints with the greatest latest known loss. Figure 1 (second row) shows the results when this recomputation (denoted by a prefix "r") is applied. It does not change the results much for $s = 10^2 \rightarrow 10^2$ and $s = 10^2 \rightarrow 1$, but drastically improves the results for $s = 10^8 \rightarrow 10^8$. The reason for this improvement is a more consistent ranking of the datapoints.

Any improvement in terms of the number of epochs to achieve certain training loss values is interesting for algorithm design. However, the performance over time is often of more practical interest,

and we show it in Figure 1 (third row). This figure demonstrates that the relative overhead coming from the batch selection procedure is rather negligible and is greatly compensated by the overall speedup of a factor of 5 observed after 5 epochs for both AdaDelta and Adam.

In order to check whether the faster convergence leads to an overfitting, we present Figure 1 (last row) with the error measured on the validation dataset over time. The results suggest that the accelerated AdaDelta and Adam achieve better validation errors faster, by a factor up to 5.

It could be possible that the obtained results are just artifacts of the used hyperparameters settings. To check this hypothesis, we performed hyperparameter optimization of both networks and learning strategies to minimize the validation error of both the original Adam and Adadelta and their variants with batch selection. Figure 2 confirms that our proposed batch selection is beneficial for training networks on MNIST and leads to a speedup of peak performance by a factor of 3. The details of the employed hyperparameter optimization algorithm (Covariance Matrix Adaptation Evolution Strategy, CMA-ES, by Hansen & Ostermeier (2001)), as well as all experimental settings, are given in the supplementary material (Section 11.5).

7 RELATED WORK

The use of uniform sampling of training examples or dual coordinates is a common way to deal with large-scale stochastic optimization problems. Although the sampled stochastic quantity is an unbiased estimate of the corresponding true quantity, the variance of the resulting stochastic quantity may be very high since individual estimates can vary significantly over different examples (Zhao & Zhang, 2014). The idea to use a non-uniform sampling is known in derivative-free optimization (more specifically, in evolutionary computation) as selection (Bäck, 1996) and is used in state-of-the-art optimizers to bias the search towards better solutions (Hansen & Ostermeier, 2001). Recently, the idea to bias the selection of more important training examples or dual coordinates has gained popularity in coordinate-descent methods (Nesterov, 2012) and stochastic gradient methods (Needell et al., 2014; Zhao & Zhang, 2014; Schmidt et al., 2015). These selection strategies are often described in the framework of importance sampling (Zhao & Zhang, 2014). They often consist of i) biasing the sampling towards the Lipschitz constants of the gradients, so that gradients that change quickly get sampled more often and gradients that change slowly get sampled less often (Schmidt et al., 2015), and ii) re-weighting the obtained gradients by inverses of their sampling probabilities to reduce the variance and unbias the estimates of gradients of the original objective function. The latter allows to make sure that all guarantees for the original and the re-weighted updates are equally valid (Needell et al., 2014).

In contrast to these works, our approach does not re-weight the gradients and does not estimate their Lipschitz constants. This allows us to render the selection to be invariant to rank-preserving transformations of the loss function. To investigate whether our approach is viable outside its original domain of application, we empirically compared it to the Non-Uniform Stochastic Average Gradient Method (SAG-NUS*) by Schmidt et al. (2015). SAG-NUS* is a state-of-the-art method which tracks estimates of Lipschitz constants to select training examples non-uniformly in order to speed up the solution of a strongly-convex optimization problem arising from the training of conditional random fields (Schmidt et al., 2015). Our batch selection approach showed competitive performance compared to SAG-NUS* (see Figure 4 in the supplementary material).

Non-uniform selection of batches of training data is virtually absent in deep learning. The only really closely relevant work we are aware of is a parallel ICLR 2016 submission by Schaul et al. (2015), where a very similar approach is used for acceleration of learning in Deep Q-Networks by prioritizing replay of relevant experiences. Another parallel ICLR submission by Alain et al. (2015) employs importance sampling for variance reduction and to obtain an unbiased estimate of the gradient in the context of parallel evaluations of gradients; however, no non-uniform selection is used. From Schaul et al. (2015), we found that Hinton (2007) used importance sampling for classification on MNIST (see also "small-data" experiments by Simpson (2015)), and, interestingly, warned that the use of extreme values of selection probabilities such as 0.01 and 100 is dangerous because "a case that used to be easy might have developed a large gradient while it was being ignored, and multiplying this gradient by 100 could give the network a shock". Note that we do not re-weight gradients, which allows us to use much larger values for the difference in probabilities s, with a best-found value in the order of 10^8. Alain et al. (2015) also notice that their algorithm's stability is in danger due to

the normalization by probabilities, the essential part of optimal importance sampling they employ (optimal in the sense of variance reduction in estimating the local gradient, but not necessarily in terms of better generalization, etc). To counter this effect, they add a small value to all probabilities, a parameter-dependent heuristic which may drift the final approach from its theoretically-sound origin. The authors of (Alain et al., 2015) note that their importance sampling approach is limited to fully connected networks (at least so far), whereas our approach directly works for any kind of network structure. We found that the use of importance sampling is not beneficial for MNIST when used with our batch selection approach (see Figure 3 in the supplementary material).

Curriculum learning is another relevant concept where the examples are not randomly presented but organized in a meaningful order which illustrates gradually more concepts, and gradually more complex ones Bengio et al. (2009). In the context of deep learning, curriculum learning can be viewed as a particular case of batch selection when $s < 1$ and not $s > 1$ as we explore in this paper.

8 LIMITATIONS

Time complexity. The sorting procedure leads to a negligible overhead for MNIST with 50.000 training datapoints. Much larger datasets will increase the absolute overhead of the sorting but the relative overhead might continue being negligible compared to the cost of learning since networks are typically much bigger then. If this is not the case, the bisection algorithm for sorting might be used instead of the current naive use of QuickSort. The overhead of the whole procedure then would scale as $O(\log N)$ per datapoint. The dominant overhead comes from the recomputation procedure, but this can be controlled by r_{freq} and r_{ratio}.

Hyperparameter settings. While the values of s in order of 10^2 - 10^8 seem reasonable for MNIST, it remains unclear what values should be used for other datasets. As for other hyperparameters, we can of course use automated hyperparameter optimization methods to set s, but we may also be able to derive a hyperparameter-less robust adaptive schedule. The simplest approach to choose s would be to set it to a small value, e.g., $s = 10$ and iteratively increase it, e.g., by a factor of 10 until the performance does not improve anymore. In this case, r_{ratio} might be set to 1 and r_{freq} (initially set to 0) should be increased. It would be of interest to automate this process similarly to the online hyperparameter adaptation procedure given in Loshchilov et al. (2014), where an internal success criterion is considered. While both AdaDelta and Adam are claimed to be robust to their hyperparameter settings, we found that their performance can be further improved by means of hyperparameter tuning (see Figure 2). However, the question of optimal hyperparameter settings is of great interest and we plan to investigate it further with CMA-ES (Hansen & Ostermeier, 2001) and standard Bayesian optimization methods (Bartz-Beielstein et al., 2005; Snoek et al., 2012; Bergstra et al., 2011; Hutter et al., 2011).

Generalization to other datasets. The main limitation of the current work is the breadth of its empirical evaluation, which only considered training of deep networks for the MNIST dataset and training of conditional random fields for optimal character recognition. The proposed method is expected to work well provided that the ranks of datapoints w.r.t. loss values are relatively stable over time. Our preliminary experiments with the CIFAR-10 dataset so far only show benefits of batch selection over random selection and not over shuffling; we are actively working on analyzing the possible causes for this and at this point only report some preliminary experiments in the supplementary material (Section 11.5).

9 CONCLUSION

While the common approach to deal with the stochasticity involved in training of deep learning networks considers a post-processing of the gradient information incoming from batches of the dataset, we propose to control the source of this information, the batch itself. More specifically, we suggest to more frequently select training datapoints with the greatest contribution to the objective function. The probability of each datapoint to be selected is decreasing with the rank of its latest known loss value. Thus, the selection itself is invariant to rank-preserving transformation of the loss function.

Our experiments demonstrated that online batch selection speeds up the convergence of the state-of-the-art DNN training methods AdaDelta and Adam by a factor of about 5. We also envision that a tighter coupling of learning approaches and batch selection would allow to provide additional speedups; for instance, one should properly link learning rates and batch size, arguably, two sides of the same coin. The increase of the batch size over time seems to be promising to deal with the noise involved in gradient estimation when approaching the optimum (see theoretical investigations by Friedlander & Schmidt (2012); Babanezhad et al. (2015) and section 11.2 in the supplementary material).

This paper provides only an initial study of possible online batch selection strategies. More advanced approaches might involve not only objective function values but also some additional indicators, such as gradient amplitudes, datapoint features/classes, and similarity to other datapoints. The selection then may consider not only the training objective function but also some proxies to the generalization capabilities of the network.

Overall, we showed that even simple batch selection mechanisms can lead to a substantial performance improvement. We hope that this finding will attract attention to this underexplored approach for speeding up the training of neural networks.

10 ACKNOWLEDGMENTS

REFERENCES

Alain, Guillaume, Lamb, Alex, Sankar, Chinnadhurai, Courville, Aaron, and Bengio, Yoshua. Variance reduction in sgd by distributed importance sampling. *arXiv preprint arXiv:1511.06481*, 2015.

Babanezhad, Reza, Ahmed, Mohamed Osama, Virani, Alim, Schmidt, Mark, Konečnỳ, Jakub, and Sallinen, Scott. Stop wasting my gradients: Practical svrg. *arXiv preprint arXiv:1511.01942*, 2015.

Bäck, Thomas. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

Bartz-Beielstein, Thomas, Lasarczyk, Christian WG, and Preuß, Mike. Sequential parameter optimization. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 1, pp. 773–780. IEEE, 2005.

Bengio, Yoshua, Louradour, Jérôme, Collobert, Ronan, and Weston, Jason. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48. ACM, 2009.

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. Algorithms for hyper-parameter optimization. In *Proc. of NIPS'11*, pp. 2546–2554, 2011.

Bordes, Antoine, Bottou, Léon, and Gallinari, Patrick. Sgd-qn: Careful quasi-newton stochastic gradient descent. *The Journal of Machine Learning Research*, 10:1737–1754, 2009.

Choromanska, Anna, Henaff, Mikael, Mathieu, Michael, Arous, Gérard Ben, and LeCun, Yann. The loss surface of multilayer networks. *arXiv preprint arXiv:1412.0233*, 2014.

Courbariaux, Matthieu, Bengio, Yoshua, and David, Jean-Pierre. Binaryconnect: Training deep neural networks with binary weights during propagations. *arXiv preprint arXiv:1511.00363*, 2015.

Dauphin, Yann N, Pascanu, Razvan, Gulcehre, Caglar, Cho, Kyunghyun, Ganguli, Surya, and Bengio, Yoshua. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in Neural Information Processing Systems*, pp. 2933–2941, 2014.

Dauphin, Yann N, de Vries, Harm, Chung, Junyoung, and Bengio, Yoshua. Rmsprop and equilibrated adaptive learning rates for non-convex optimization. *arXiv preprint arXiv:1502.04390*, 2015.

Deng, L., Hinton, G., and Kingsbury, B. New types of deep neural network learning for speech recognition and related applications: An overview. In *Proc. of ICASSP'13*, 2013.

Donahue, J., Jia, Y., Vinyals, O., Hoffman, J., Zhang, N., Tzeng, E., and Darrell, T. Decaf: A deep convolutional activation feature for generic visual recognition. In *Proc. of ICML'14*, 2014.

Duchi, John, Hazan, Elad, and Singer, Yoram. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

Fletcher, Roger. A new approach to variable metric algorithms. *The computer journal*, 13(3): 317–322, 1970.

Friedlander, Michael P and Schmidt, Mark. Hybrid deterministic-stochastic methods for data fitting. *SIAM Journal on Scientific Computing*, 34(3):A1380–A1405, 2012.

Fukumizu, Kenji and Amari, Shun-ichi. Local minima and plateaus in hierarchical structures of multilayer perceptrons. *Neural Networks*, 13(3):317–327, 2000.

Hansen, Nikolaus and Ostermeier, Andreas. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

Hansen, Nikolaus and Ros, Raymond. Benchmarking a weighted negative covariance matrix update on the bbob-2010 noiseless testbed. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*, pp. 1673–1680. ACM, 2010.

Hazan, Elad, Levy, Kfir Y, and Shalev-Shwartz, Shai. Beyond convexity: Stochastic quasi-convex optimization. *arXiv preprint arXiv:1507.02030*, 2015.

Hinton, Geoffrey E. To recognize shapes, first learn to generate images. *Progress in brain research*, 165:535–547, 2007.

Hutter, F., Hoos, H., and Leyton-Brown, K. Sequential model-based optimization for general algorithm configuration. In *Proc. of LION'11*, pp. 507–523, 2011.

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Kingma, Diederik and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Krizhevsky, A., Sutskever, I., and Hinton, G. Imagenet classification with deep convolutional neural networks. In *Proc. of NIPS'12*, pp. 1097–1105, 2012.

LeCun, Yann, Bottou, Léon, Bengio, Yoshua, and Haffner, Patrick. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

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

Loshchilov, Ilya, Schoenauer, Marc, Sebag, Michele, and Hansen, Nikolaus. Maximum likelihood-based online adaptation of hyper-parameters in CMA-ES. In *Parallel Problem Solving from Nature–PPSN XIII*, pp. 70–79. Springer, 2014.

Needell, Deanna, Ward, Rachel, and Srebro, Nati. Stochastic gradient descent, weighted sampling, and the randomized kaczmarz algorithm. In *Advances in Neural Information Processing Systems*, pp. 1017–1025, 2014.

Nesterov, Yu. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 22(2):341–362, 2012.

Roller, Ben Taskar Carlos Guestrin Daphne. Max-margin markov networks. *Advances in neural information processing systems*, 16:25, 2004.

Schaul, Tom, Zhang, Sixin, and LeCun, Yann. No more pesky learning rates. *arXiv preprint arXiv:1206.1106*, 2012.

Schaul, Tom, Quan, John, Antonoglou, Ioannis, and Silver, David. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

Schmidt, Mark, Babanezhad, Reza, Ahmed, Mohamed Osama, Defazio, Aaron, Clifton, Ann, and Sarkar, Anoop. Non-uniform stochastic average gradient method for training conditional random fields. *arXiv preprint arXiv:1504.04406*, 2015.

Simpson, Andrew JR. " oddball sgd": Novelty driven stochastic gradient descent for training deep neural networks. *arXiv preprint arXiv:1509.05765*, 2015.

Snoek, J., Larochelle, H., and Adams, R. P. Practical Bayesian optimization of machine learning algorithms. In *Proc. of NIPS'12*, pp. 2960–2968, 2012.

Sutton, Richard S and Barto, Andrew G. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

Zeiler, Matthew D. Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

Zhao, Peilin and Zhang, Tong. Stochastic optimization with importance sampling. *arXiv preprint arXiv:1401.2753*, 2014.

11 SUPPLEMENTARY MATERIAL

11.1 WHEN $N_{eff} = N$?

The selection pressure parameter s affects the effective size of the training data we used. Let N_{eff} denote the number of unique datapoints selected in the last N steps; then, N_{eff} is likely to decrease when s increases. However, this is not necessarily the case: albeit unlikely, one possible scenario in which N_{eff} would still equal N is the following: if $p_1 = \infty$ and $p_{i>1} = 0$, the datapoint with the greatest latest known loss $f_t(\boldsymbol{x}_t)$ is selected and the optimization algorithm $\mathcal{A}$ uses $\nabla f_t(\boldsymbol{x}_t)$ to minimize $f_t(\boldsymbol{x}_t)$, the new value $f_t(\boldsymbol{x}_t)$ appears to be the lowest loss value seen so far, the datapoint is ranked last and its selection is then scheduled to be in N steps, the whole situation happens again and again leading to the sweep over the whole dataset and thus $N_{eff} = N$.

11.2 ONLINE SELECTION OF THE BATCH SIZE

We decided to move the following material to simplify the main paper.

Not only the content of the selected batch, but also its size represents a design parameter of interest. When $b = N$, we have a denoised access to f. When $b = 1$, we obtain a noisy but N times cheaper access to f. The negative impact of this noise supposedly depends on the distance to the optimum in the objective and/or decision space. As we progress to the optimum over time, increasing the batch size is an approach to deal with the noise. Moreover, the use of a relatively small batch size in the beginning of search has a potential to bring a significant speedup, however, it has a potential risk of performing unnecessary random walk. We therefore propose to increase batch size over time. More specifically, we propose to consider both exponential and linear increases of b over epochs e:

$$b_e = b_{e_0} \exp(\log(b_{e_{end}}/b_{e_0})/(e_{end} - e_0))^{e_{end}-e_0}, \tag{6}$$

$$b_e = b_{e_0} + (b_{e_{end}} - b_{e_0})(e - e_0)/e_{end}, \tag{7}$$

where e_0 and e_{end} correspond to the first and last epochs of the increase (not necessarily the last epoch of optimization).

The closer approach of the optimum requires a greater batch size to be used to overcome the imprecision of gradient estimation. Therefore, starting with relatively small batches on easy training/optimization problems does not seem to be a bad idea. This intuition is supported by the recent results of Hazan et al. showing that the Normalized Stochastic Gradient Descent (NSGD) (when the gradient is normalized by its amplitude) provably requires a minimum batch size to achieve an ϵ precision, without which the convergence of the NSGD cannot be guaranteed. The detailed theoretical analysis of advantages of batch size scheduling can be found in (Friedlander & Schmidt, 2012; Babanezhad et al., 2015).

We do not report the results for this section because the additional speedup provided by this procedure is pronounced in terms of epochs but not in terms of time. This is the case because small batches are more expensive (per datapoint) to compute on a GPU. However, we do include batch size selection procedure when dealing with hyper-parameter optimization (see section 11.5).

11.3 IMPORTANCE SAMPLING

We use Importance Sampling (IS) on top of our batch selection approach. More specifically, we normalized gradients by normalized probabilities $p_i / \sum_{i=1}^{b} p_i$ of the corresponding datapoint to be selected (Zhao & Zhang, 2014). Figure 3 shows the results of our original approach (solid lines) and its modified version (dotted lines) where IS is used. The latter shows a worse validation accuracy in all cases.

11.4 RANK-BASED SELECTION FOR NON-UNIFORM STOCHASTIC AVERAGE GRADIENT METHOD

The Non-Uniform Stochastic Average Gradient Method (SAG-NUS*) is a state-of-the-art method which tracks Lipschitz constants estimates to select training examples for faster solving a strongly-
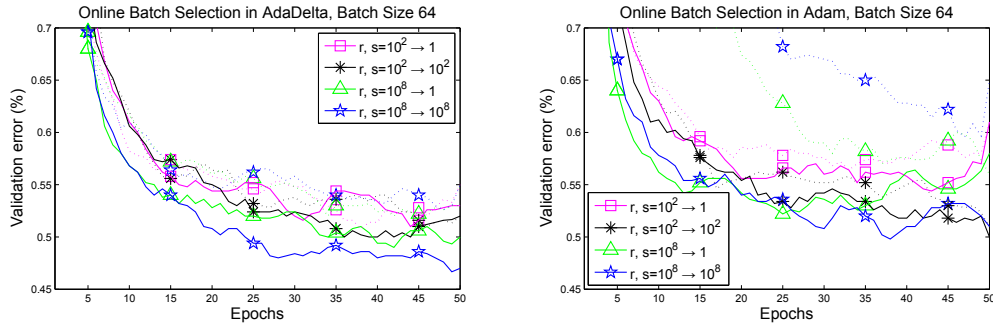
Figure 3: Convergence curves of AdaDelta (**Left**) and Adam (**Right**) on MNIST dataset. Bold lines correspond to our original approach, dotted lines correspond to its version with Importance Sampling. The value of s denotes the ratio of probabilities to select the training datapoint with the greatest latest known loss rather than the smallest latest known loss. Legends with $s = s_{e_0} \to s_{e_{end}}$ correspond to an exponential change of s from s_{e_0} to $s_{e_{end}}$ as a function of epoch index e, see Eq. (5). Legends with the prefix "r" correspond to the case when the loss values for $r_{ratio} \cdot N = 1.0N$ datapoints with the greatest latest known loss are recomputed $r_{freq} = 0.5$ times per epoch. All curves are computed on the whole training set and smoothed by a moving average of the median of 11 runs.
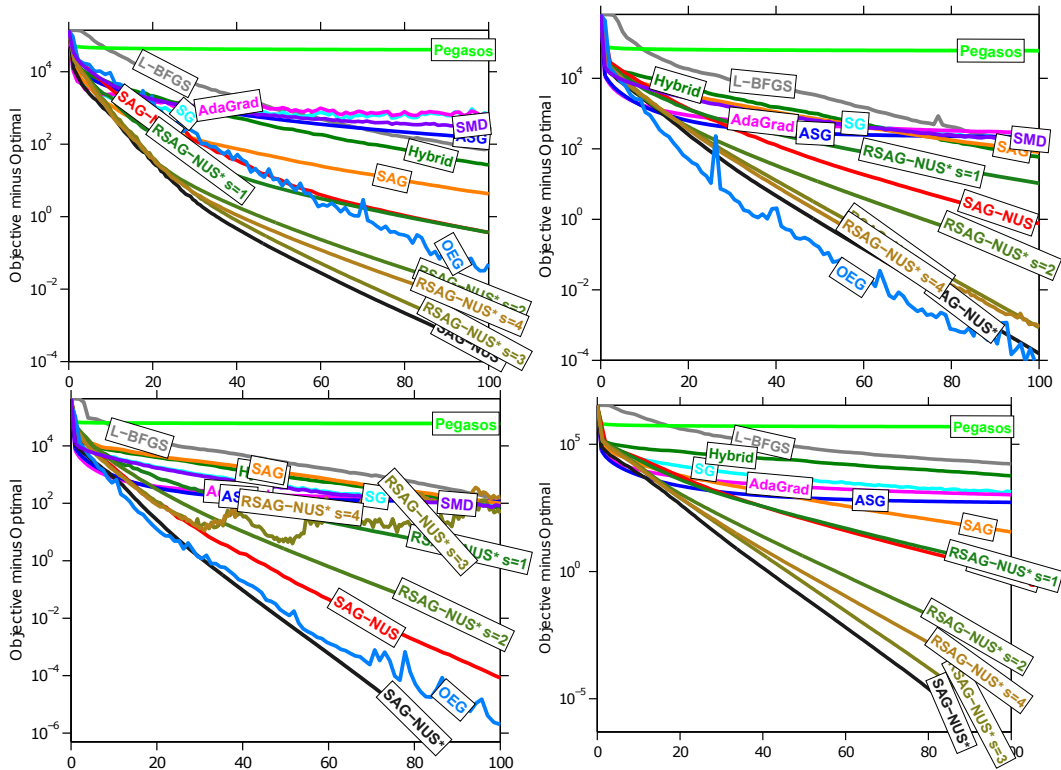

Figure 4: Objective minus optimal objective value against effective number of passes for different deterministic, stochastic, and semi-stochastic optimization strategies. Top-left: OCR, Top-right: CoNLL-2000, bottom-left: CoNLL-2002, bottom-right: POS-WSJ.

convex optimization problem arising from the training of Conditional Random Fields (CRFs) (Schmidt et al., 2015). We replaced the Lipschitz constants-based proportional selection by our rank-based selection which does not depend of Lipschitz constants themself but only on their ranks; the rest of SAG-NUS* remained unchanged. We benchmark a wide variety of approaches using the

source code kindly provided by Schmidt et al. (2015) on four CRF training tasks: the optical character recognition (OCR) dataset of Roller (2004), the CoNLL-2000 shallow parse chunking dataset [2], the CoNLL-2002 Dutch named-entity recognition dataset [3], and a part-of-speech (POS) tagging task using the Penn Treebank Wall Street Journal data (POS-WSJ). The proper description of all algorithms and experimental settings is given in (Schmidt et al., 2015). We denote our modification of SAG-NUS* with rank-based selection as rSAG-NUS* and parameterize it by s denoting the ratio between the greatest and smallest selection probabilities in the same way as in the main paper. We did not try to refresh the ranking as in the main paper.

The results of rSAG-NUS* with $s = 1$ showed in Figure 4 corresponds to uniform selection and defines a baseline. The results for rSAG-NUS* with $s = 3$ are competitive to SAG-NUS* on three out of four datasets. The reason of a divergence seen for the CoNLL-2002 and rSAG-NUS* with $s = 3, s = 4$ is not yet clear but the recomputation procedure (which we omitted in these experiments) might help to avoid it. This preliminary experiment suggests that our rank-based selection is a viable alternative (in three cases out of four) to the Lipschitz proportional selection. The source code to reproduce the experiments and Figure 4 is available at `https://sites.google.com/site/batchsel/`.

Table 1: Hyperparameters descriptions, pseudocode transformations and ranges

name	description	transformation	range
x_1	selection pressure at e_0	$10^{-2+10^{2x_1}}$	$[10^{-2}, 10^{98}]$
x_2	selection pressure at e_{end}	$10^{-2+10^{2x_2}}$	$[10^{-2}, 10^{98}]$
x_3	batch size at e_0	2^{4+4x_3}	$[2^4, 2^8]$
x_4	batch size at e_{end}	2^{4+4x_4}	$[2^4, 2^8]$
x_5	frequency of loss recomputation r_{freq}	$2x_5$	$[0, 2]$
x_6	alpha for batch normalization	$0.01 + 0.2x_6$	$[0.01, 0.21]$
x_7	epsilon for batch normalization	10^{-8+5x_7}	$[10^{-8}, 10^{-3}]$
x_8	dropout rate after the first Max-Pooling layer	$0.8x_8$	$[0, 0.8]$
x_9	dropout rate after the second Max-Pooling layer	$0.8x_9$	$[0, 0.8]$
x_{10}	dropout rate before the output layer	$0.8x_{10}$	$[0, 0.8]$
x_{11}	number of filters in the first convolution layer	$2^{3+5x_{11}}$	$[2^3, 2^8]$
x_{12}	number of filters in the second convolution layer	$2^{3+5x_{12}}$	$[2^3, 2^8]$
x_{13}	number of units in the fully-connected layer	$2^{4+5x_{13}}$	$[2^4, 2^9]$
x_{14}	Adadelta: learning rate at e_0	$10^{0.5-2x_{14}}$	$[10^{-1.5}, 10^{0.5}]$
x_{15}	Adadelta: learning rate at e_{end}	$10^{0.5-2x_{15}}$	$[10^{-1.5}, 10^{0.5}]$
x_{16}	Adadelta: ρ	$0.8 + 0.199x_{16}$	$[0.8, 0.999]$
x_{17}	Adadelta: ϵ	$10^{-3-6x_{17}}$	$[10^{-9}, 10^{-3}]$
x_{14}	Adam: learning rate at e_0	$10^{-1-3x_{14}}$	$[10^{-4}, 10^{-1}]$
x_{15}	Adam: learning rate at e_{end}	$10^{-3-3x_{15}}$	$[10^{-6}, 10^{-3}]$
x_{16}	Adam: β_1	$0.8 + 0.199x_{16}$	$[0.8, 0.999]$
x_{17}	Adam: ϵ	$10^{-3-6x_{17}}$	$[10^{-9}, 10^{-3}]$
x_{18}	Adam: β_2	$1 - 10^{-2-2x_{18}}$	$[0.99, 0.9999]$
x_{19}	adaptation end epoch index e_{end}	$20 + 200x_{19}$	$[20, 220]$

11.5 HYPERPARAMETER OPTIMIZATION

We investigated whether the use batch selection is still beneficial when hyperparameters are optimized. We used Covariance Matrix Adaptation Evolution Strategy (CMA-ES) by Hansen & Ostermeier (2001) in its "active" regime (Hansen & Ros, 2010) as implemented in a Python code available at `https://pypi.python.org/pypi/cma`. We used a greater than default population size $\lambda = 30$ to benefit from parallel evaluations of solutions on a mixed set of 30 GPU's, mostly GeForce GTX TITAN Black. Evaluation of each solution involves network building and training using input hyperparameter values. The total time for each evaluation is limited. The best validation error found in all epochs represents the objective function. Table 1 shows the names of hyperparameters, their

---

[2] `http://www.cnts.ua.ac.be/conll2000/chunking`
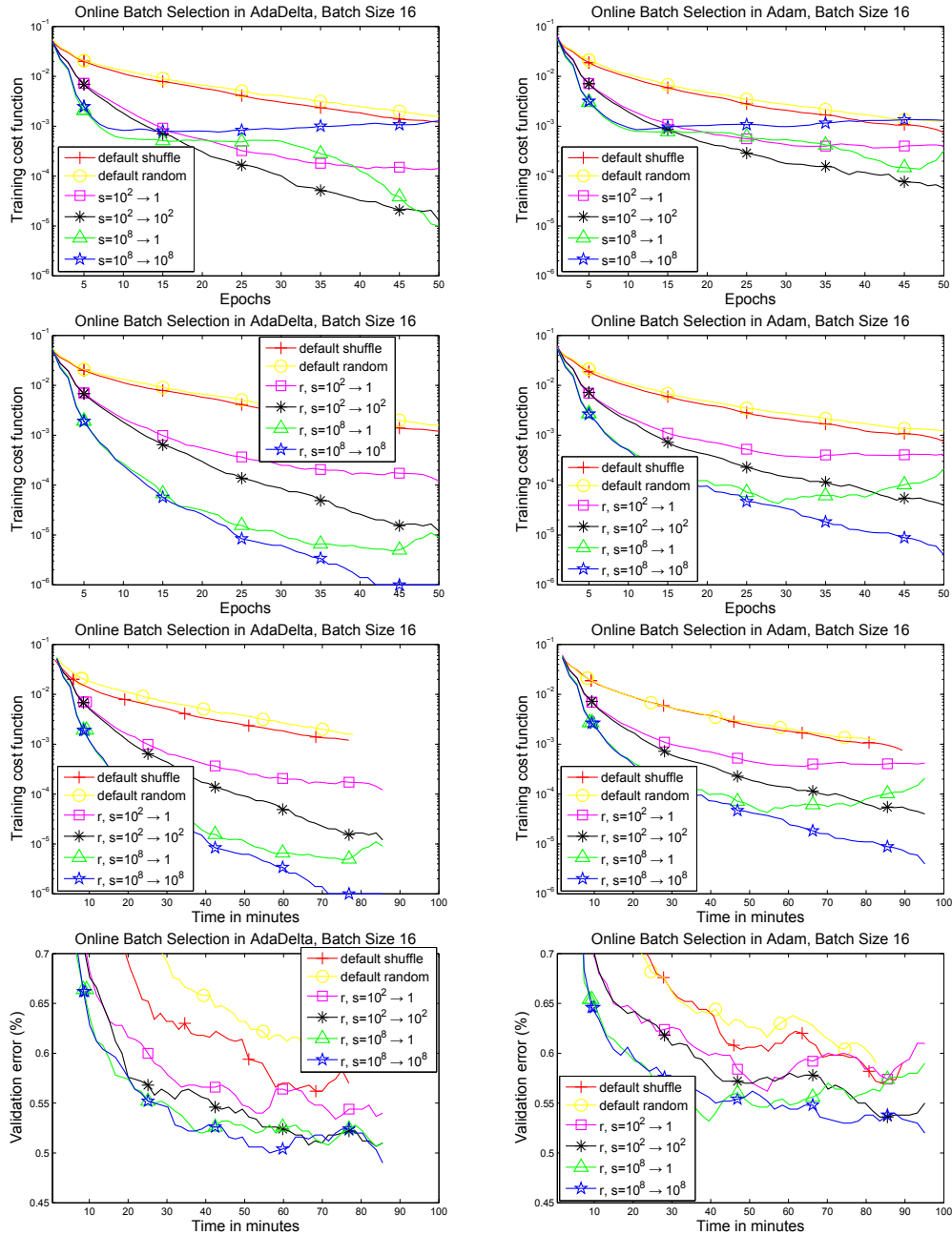[3] `http://www.cnts.ua.ac.be/conll2002/ner`

Figure 5: Convergence curves of AdaDelta (**Left Column**) and Adam (**Right Column**) on MNIST dataset. The original algorithms are denoted by "default shuffle" (respectively, "default with random") when datapoints are shuffled and selected sequentially (respectively, uniformly at random). The value of s denotes the ratio of probabilities to select the training datapoint with the greatest latest known loss rather than the smallest latest known loss. Legends with $s = s_{e_0} \rightarrow s_{e_{end}}$ correspond to an exponential change of s from s_{e_0} to $s_{e_{end}}$ as a function of epoch index e, see Eq. (5). Legends with the prefix "r" correspond to the case when the loss values for $r_{ratio} \cdot N = 1.0N$ datapoints with the greatest latest known loss are recomputed $r_{freq} = 0.5$ times per epoch. All curves are computed on the whole training set and smoothed by a moving average of the median of 11 runs.
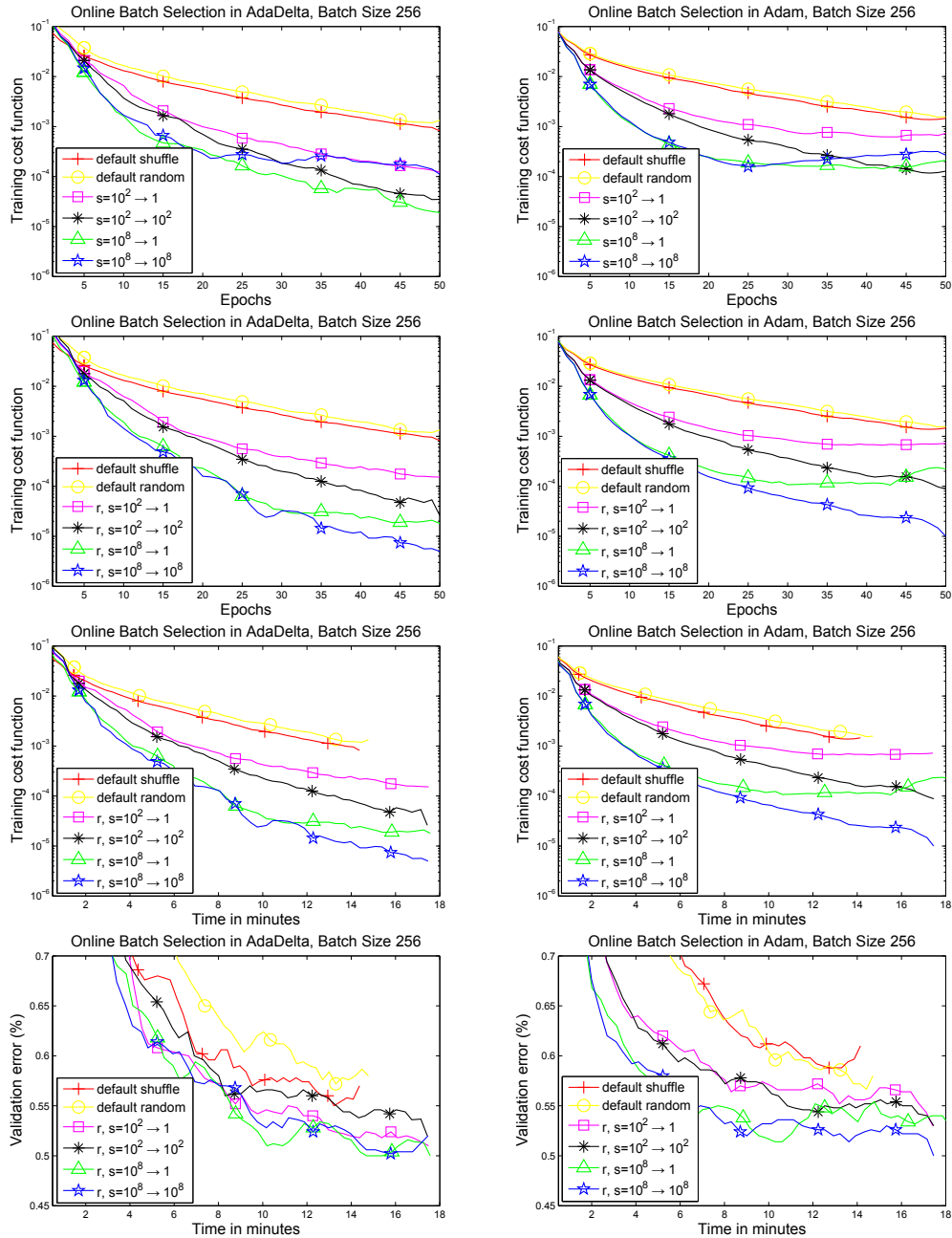
Figure 6: Convergence curves of AdaDelta (**Left Column**) and Adam (**Right Column**) on MNIST dataset. The original algorithms are denoted by "default shuffle" (respectively, "default with random") when datapoints are shuffled and selected sequentially (respectively, uniformly at random). The value of s denotes the ratio of probabilities to select the training datapoint with the greatest latest known loss rather than the smallest latest known loss. Legends with $s = s_{e_0} \rightarrow s_{e_{end}}$ correspond to an exponential change of s from s_{e_0} to $s_{e_{end}}$ as a function of epoch index e, see Eq. (5). Legends with the prefix "r" correspond to the case when the loss values for $r_{ratio} \cdot N = 1.0N$ datapoints with the greatest latest known loss are recomputed $r_{freq} = 0.5$ times per epoch. All curves are computed on the whole training set and smoothed by a moving average of the median of 11 runs.

description and the way to transform the initial variables from the ranges $[0, 1]$ (note, we constrain the search) to the actual ranges. Some variables take different ranges depending on whether we run Adadelta or Adam, the others remain inactive (e.g., x_1, x_2, x_5) when batch selection is not used.

We performed one run for Adadelta, Adam, Adadelta with batch selection and Adam with batch selection for time budgets of 5, 10, 15 and 30 minutes. The mean of CMA-ES was set to 0.5 for all variables, the initial step-size was set to 0.2. The results for 34 iterations, about 1000 function evaluations, are given in Figure 7. The results suggest that batch selection is beneficial both when hyperparameters are random (in the very beginning of the run) and optimized (in the end of the run). It can be seen that the results found for Adadelta and Adam with batch selection and time budget of 10 minutes are similar to the ones found by the original Adadelta and Adam after 30 minutes. This suggests that the speedup of a factor of 3 is observed (see also Figure 2).

Figure 8 shows the results of our preliminary attempt to apply batch selection for the CIFAR-10 dataset, in a similar way as Figure 2 in the main paper does for the MNIST dataset. We note that in this very preliminary experiment on CIFAR-10, batch selection performed slightly better than random selection, but not yet better than shuffling; we are currently analyzing the reasons for this behavior and are extending batch selection to generalize shuffling. We also note that, for a network training time of 60 minutes, CMA-ES was able to improve validation error of the best solution found from 12% in the first generation to about 10% (found at the latest 34-th generation; we used population size 15 on 15 GeForce GTX TITAN X).

To our best knowledge, our experiments are the first where CMA-ES is used for optimization of hyperparameters of deep neural networks. A more detailed analysis and comparison with other techniques is required.
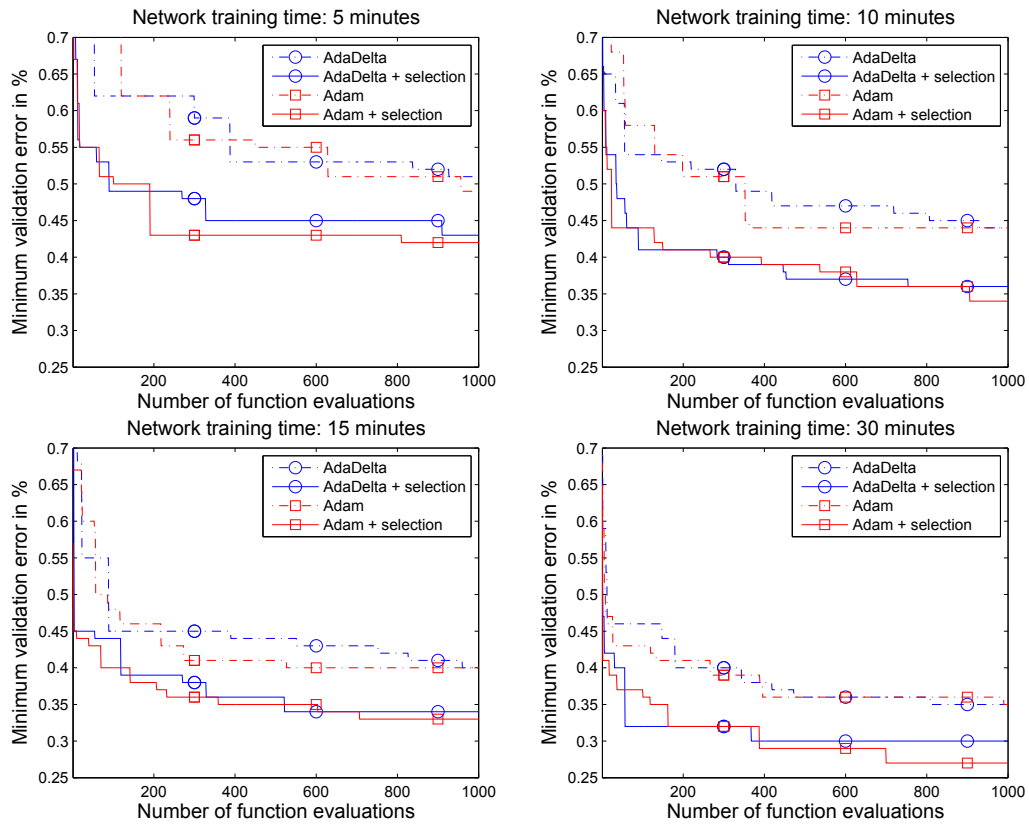
Figure 7: Best validation errors found for AdaDelta and Adam with and without batch selection when hyperparameters are optimized by CMA-ES. The results are given for different training time budgets: 5, 10, 15 and 30 minutes.
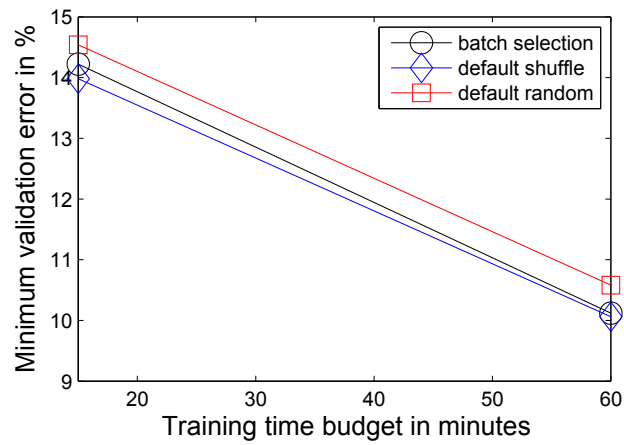
Figure 8: Minimum validation error found on CIFAR-10 for deep neural network trained by Adam with a given budget of time (15 and 60 minutes). The results correspond to the best hyperparameter settings found by CMA-ES with population size 15 after 510 function evaluations.