Preprints of the 21st IFAC World Congress (Virtual)
Berlin, Germany, July 12-17, 2020

An Adaptive Memory Multi-Batch L-BFGS Algorithm for Neural Network Training

Federico Zocco [⋆] **Seán McLoone**

Centre for Intelligent Autonomous Manufacturing Systems, Queen's University Belfast, Northern Ireland, UK (e-mail: {fzocco01, s.mcloone}@qub.ac.uk)

Abstract: Motivated by the potential for parallel implementation of batch-based algorithms and the accelerated convergence achievable with approximated second order information a limited memory version of the BFGS algorithm has been receiving increasing attention in recent years for large neural network training problems. As the shape of the cost function is generally not quadratic and only becomes approximately quadratic in the vicinity of a minimum, the use of second order information by L-BFGS can be unreliable during the initial phase of training, i.e. when far from a minimum. Therefore, to control the influence of second order information as training progresses, we propose a multi-batch L-BFGS algorithm, namely MB-AM, that gradually increases its trust in the curvature information by implementing a progressive storage and use of curvature data through a development-based increase (*dev-increase*) scheme. Using six discriminative modelling benchmark problems we show empirically that MB-AM has slightly faster convergence and, on average, achieves better solutions than the standard multi-batch L-BFGS algorithm when training MLP and CNN models.

Keywords: Deep learning, L-BFGS, variable memory, quasi-Newton methods, neural networks

1. INTRODUCTION

In the last twenty years significant advances have been made towards making artificial neural networks able to compete with their biological counterparts (Dodge and Karam (2017)). A factor that is critical to the performance of artificial neural networks is the network training algorithm, which determines the sequence of computations to be performed in order for the network to learn the underlying relationships in the dataset of interest. Many training algorithms have been proposed over the years to achieve this goal, typically trading off computational complexity with rate-of-convergence and/or quality of solutions obtained (McLoone et al. (1998); Goodfellow et al. (2016); Ruder (2016)). In general, training algorithms fall into three categories: first order methods, which calculate the loss function and its derivative at each iteration, e.g. stochastic gradient descent (SGD) and Adam (Kingma and Ba (2014)), second order methods, which also calculate second derivative information, and quasi-Newton methods, which evaluate an approximation of the second derivative instead of computing it directly, e.g. the limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) (Liu and Nocedal (1989)). Whenever the amount of data to be processed is large, state-of-the-art performance is usually achieved with well-tuned first order methods thanks to regularization techniques such as batch normalization and dropout (Bollapragada et al. (2018)). Regularization and stable approximated curvature updates in L-BFGS methods are currently an active area of research due to the accelerated convergence achievable with curvature information and the ability to exploit parallelism with large batch sizes to achieve efficient algorithm implementations (Berahas et al. (2016); Yousefian et al. (2017)). Agarwal et al. (2014) proposed a hybrid approach where training is initially performed with a fast and regularized first order method and then switched to a full batch method to exploit parallelism. The L-BFGS method we propose can be seen as a hybrid approach where the batch size is kept fixed as in Berahas et al. (2016), but the use of approximated second order information is gradually increased during training thereby enhancing convergence. Initially, when little second order information is used, the method is essentially first order and becomes increasingly second order in nature as more and more curvature information is incorporated into the weight updates.

Our contributions: Motivated by recent interest in progressive training strategies (Smith et al. (2017); Bollapragada et al. (2018)), we propose an approach based on *progressive curvature trust* for the multi-batch L-BFGS algorithm of Berahas et al. (2016) by adding two algorithmic ingredients: a progressive storage (and use) of curvature information, and periodic resetting. Based on the development-based decay (*dev-decay*) scheme for first order methods (Wilson et al. (2017)), we propose a *dev-increase* scheme to control the progression of the memory size for curvature information and its resetting. For comparison purposes, we develop three variants of the multi-batch L-BFGS and experimentally compare them with the multi-batch L-BFGS and Adam algorithms for training multilayer perceptrons (MLPs) and convolutional neural networks (CNNs) on six case study datasets.

⋆ The first author gratefully acknowledges the financial support provided by Irish Manufacturing Research (IMR) for this research.

The paper is organized as follows: Section 2 describes the multi-batch L-BFGS algorithm of Berahas et al. (2016) as this is the reference algorithm to which we add two algorithmic ingredients; Section 3 explains the concept of *progressive curvature trust* and the proposed algorithm; Section 4 describes the case studies and experiments conducted to evaluate the proposed algorithm and discusses the results obtained. Finally, Section 5 gives the conclusions.

Hereinafter, matrices and vectors are indicated with bold capital and bold lowercase letters, while lowercase italic font denotes scalars. The sets are indicated by italic capital letters.

2. MULTI-BATCH L-BFGS

Let us consider a classification task defined by the set

$$\mathcal{B} = \{\boldsymbol{x}^{(i)}, l^{(i)}\}_{i=1,2,\ldots,|\mathcal{B}|},$$

where $\boldsymbol{x}^{(i)} \in \mathbb{R}^n$ is an input sample (e.g. an image) of size n and $l^{(i)} \in \mathbb{N}$ is an integer representing its class (i.e. label). A neural network relates the input to the output label via a mapping $p(\boldsymbol{\theta}, \boldsymbol{x}^{(i)}) : \mathbb{R}^n \mapsto \mathbb{N}$, with network parameters $\boldsymbol{\theta} \in \mathbb{R}^d$ optimally adapted to $\mathcal{B}$ using a training algorithm. Full-batch algorithms learn $\boldsymbol{\theta}$ by considering the complete dataset $\mathcal{B}$, therefore they optimize a deterministic function $C(\mathcal{B})$, that is

$$\min_{\boldsymbol{\theta}} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} c(\boldsymbol{x}^{(i)}, l^{(i)}; p(\boldsymbol{\theta}, \boldsymbol{x}^{(i)})) = \min_{\boldsymbol{\theta}} C(\mathcal{B}),$$

where $c(\cdot)$ is a cost (i.e. loss) function chosen to measure the distance between the model prediction $\hat{l}^{(i)} = p(\boldsymbol{\theta}, \boldsymbol{x}^{(i)})$ and the expected output $l^{(i)}$ when given the sample $\boldsymbol{x}^{(i)}$ as input. In contrast, in multi-batch mode a random subset $\mathcal{S}_k \subset \mathcal{B}$ is used at each iteration and the optimal $\boldsymbol{\theta}$ is estimated by iteratively minimizing a stochastic cost function $C(\mathcal{S}_k)$, that is

$$\min_{\boldsymbol{\theta}_k} \frac{1}{|\mathcal{S}_k|} \sum_{i \in \mathcal{S}_k} c(\boldsymbol{x}^{(i)}, l^{(i)}; p(\boldsymbol{\theta}, \boldsymbol{x}^{(i)})) = \min_{\boldsymbol{\theta}_k} C(\mathcal{S}_k), \tag{1}$$

where k is the iteration counter. The L-BFGS algorithm updates the parameters with the rule

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \boldsymbol{H}_k^{-1} \boldsymbol{g}_k^{\mathcal{S}_k} \tag{2}$$

where η is the learning rate (i.e. step length), $\boldsymbol{g}_k^{\mathcal{S}_k}$ is the gradient defined as

$$\boldsymbol{g}_k^{\mathcal{S}_k} = \frac{\partial C(\mathcal{S}_k)}{\partial \boldsymbol{\theta}_k} \tag{3}$$

and $\boldsymbol{H}_k^{-1}$ is the inverse Hessian matrix approximation updated according to (Nocedal and Wright (2006))

$$\begin{aligned}
\boldsymbol{H}_{k+1}^{-1} &= \boldsymbol{V}_k^\top \boldsymbol{H}_k^{-1} \boldsymbol{V}_k + \rho_k \boldsymbol{s}_k \boldsymbol{s}_k^\top \\
\rho_k &= (\boldsymbol{t}_k^\top \boldsymbol{s}_k)^{-1} \\
\boldsymbol{V}_k &= \boldsymbol{I} - \rho_k \boldsymbol{t}_k \boldsymbol{s}_k^\top \\
\boldsymbol{s}_k &= \boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k && (4) \\
\boldsymbol{t}_k &= \boldsymbol{g}_{k+1}^{\mathcal{S}_k} - \boldsymbol{g}_k^{\mathcal{S}_k}. && (5)
\end{aligned}$$

The storage of $\boldsymbol{H}_k^{-1}$ requires $\mathcal{O}(d^2)$ memory, which is prohibitive with large neural networks, therefore in L-BFGS a two-loop recursion is used to directly compute the product $\boldsymbol{H}_k^{-1} \boldsymbol{g}_k^{\mathcal{S}_k}$ such that only a predefined number of curvature pairs $(\boldsymbol{s}_k, \boldsymbol{t}_k)$ need to be stored. The maximum number of stored pairs, m, is usually fixed between 3 and 20 (Nocedal and Wright (2006)). Once the limit m is reached the oldest pairs are replaced by the newest ones. Given the stochastic nature of the evaluation of the gradient using (3), to achieve a stable computation of the curvature $\boldsymbol{t}_k$ (see (5)) Berahas et al. (2016) proposed having overlapping consecutive batches, i.e. $\mathcal{O}_k = \mathcal{S}_{k-1} \cap \mathcal{S}_k \neq \emptyset$. This overlap is defined with the hyperparameter $o = \frac{|\mathcal{O}_k|}{|\mathcal{S}_k|}$, which is usually chosen in the range $0 < o < 0.5$.

3. ADAPTIVE MEMORY MULTI-BATCH L-BFGS

3.1 Local Approximation of the Cost Function

An important feature of gradient-based optimization algorithms is the computation of the search direction used to find an updated parameter vector $\boldsymbol{\theta}_{k+1}$. Newton and quasi-Newton methods such as L-BFGS define the direction approximating the exact cost function C with a quadratic model, obtained by truncating the cost function Taylor series expansion at the third term, that is:

$$\begin{aligned}
C(\boldsymbol{\theta}_k + \boldsymbol{y}_k) \approx \bar{C}_k = \\
= C(\boldsymbol{\theta}_k) + \boldsymbol{g}_k^\top(\boldsymbol{\theta}_k)\boldsymbol{y}_k + \frac{1}{2}\boldsymbol{y}_k^\top \boldsymbol{H}_k(\boldsymbol{\theta}_k)\boldsymbol{y}_k.
\end{aligned}$$

Then, differentiating with respect to $\boldsymbol{y}_k$ and setting it equal to zero we find the direction leading to the minimum of $\bar{C}$

$$\boldsymbol{y}_k = -\boldsymbol{H}_k^{-1} \boldsymbol{g}_k,$$

which motivates the weight update rule in (2). However, if around the point $\boldsymbol{\theta}_k + \boldsymbol{y}_k$ the function is significantly different from a quadratic-like model, the use of the curvature information could be detrimental to stable and reliable parameter updates. Moreover the curvature evaluation is computationally demanding as it requires the computation of $d(d+1)/2$ derivatives, thus it is only worth using if it actually improves the algorithm convergence.

Now we observe that as we approach a minimum point of a nonlinear multi-modal function the quadratic approximation becomes increasingly valid, and this occurs at the more advanced stages of training, i.e.

$$\lim_{k \to \infty} C - \bar{C}_k = 0.$$

In contrast, at the beginning of the training process it is difficult to say how well a quadratic model will approximate the true cost function.

Therefore, we propose an algorithm that implements a progressive use of curvature information: initially only a few curvature pairs (e.g. one or two) are stored and used, with the older pairs discarded to reduce the computational cost; then, when the algorithm gets closer to the minimum, we increase the storage and use of second order information.

3.2 The Algorithm

The pseudo-code of the proposed method is presented in Algorithm 1, which is the result of the combination of three building blocks:

Block 1: The basic structure is the multi-batch L-BFGS of Berahas et al. (2016) described in Section 2 which performs stable curvature updates by overlapping consecutive batches. A PyTorch implementation is available on GitHub [1].

Block 2: To regulate the use of approximated second order information during training the memory m_k used to store the curvature pairs is increased as the number of iterations increases, in a fashion similar to the progressive batching approach proposed in Bollapragada et al. (2018). The rule defining when to increase m_k is based on the development of the validation loss in a similar fashion to the *dev-decay* scheme of Wilson et al. (2017) applied to the step length. Adapted to implement a progressive curvature trust, it becomes a *dev-increase* scheme. It is executed between Step 11 and Step 18, with α denoting the memory scaling factor. The macro-condition (6) is typically satisfied when the validation loss is approaching a local minimum. The parameter m_{val} defines how many previous validation losses are stored and, if increased, makes the satisfaction of Q more difficult, thus delaying the increase of m_k. The lower and upper limits of m_k are m_0 and m_{max}, respectively, i.e. $m_0 \leq m_k \leq m_{max}$. Note that the extra memory required to store the validation losses is negligible with large datasets/models because v_k is a scalar and m_{val} is a relatively small number (e.g. $m_{val} \leq 10$).

$$Q : \{\Delta_{k-m_{val}+j} > \Delta_{k-m_{val}+j+1}\}_{j=0,1,\dots,m_{val}-2} \qquad (6)$$

with

$$\Delta_k = v_{k-1} - v_k.$$

Block 3: To further mitigate the impact of using unrepresentative curvature information, the resetting of the memory proposed in McLoone and Irwin (1999) is implemented. Adapted to implement a progressive curvature trust, the resetting is applied just for smaller k according to the development of m_k, which in turn follows the validation loss development. This third building block is implemented between Step 19 and Step 21, where q_k is the current number of stored curvature pairs and m_{reset} is the memory size below which resetting is applied.

4. EXPERIMENTS

4.1 Experimental Setup

In this section two implementations based on Algorithm 1 are considered: multi-batch L-BFGS with adaptive memory and without resetting (MB-AM), i.e. $m_{reset} = 0$, and multi-batch L-BFGS with both adaptive memory and resetting (MB-AMR), i.e. $m_{reset} > 0$. In addition, a multi-batch L-BFGS with constant memory and periodic resetting, as proposed in McLoone and Irwin (1999), is considered. This variant, denoted as MB-R, performs resetting whenever the maximum memory size is reached. For benchmarking purposes, two additional methods are considered, namely, standard multi-batch L-BFGS (MB), and Adam. Therefore in total five methods are compared.

The six datasets and six experiments considered are detailed in Table 1 and Table 2, respectively. The MLP is a single-hidden layer fully connected feedforward neural network with h neurons in the hidden layer. The CNN has two 2D convolutional layers followed by two fully connected layers. Each convolutional layer is followed by

Algorithm 1 Adaptive Memory Multi-Batch L-BFGS

```
 1: Input: $\boldsymbol{\theta}_0$ (initial iterate), $\mathcal{B} = \{(\boldsymbol{x}^{(i)}, l^{(i)}), \text{ for } i \in \mathcal{B}\}$
    (training set), $r$ (i.e. $|\mathcal{S}_k|$), $o$ (overlap ratio), $\eta$, $\alpha$, $m_0$,
    $m_{max}$, $m_{val}$, $m_{reset}$
 2: Initialisation: $k \leftarrow 0$, $q_k \leftarrow 0$, $m_k \leftarrow m_0$
 3: Randomly select a batch $\mathcal{S}_0 \subset \mathcal{B}$
 4: Repeat until convergence
 5: Compute $C_k$
 6: Compute $\boldsymbol{g}_k^{\mathcal{S}_k}$
 7: Compute $\boldsymbol{H}_k \boldsymbol{g}_k^{\mathcal{S}_k}$ via L-BFGS two-loop recursion (No-
    cedal and Wright (2006))
 8: Update the parameters as in (2)
 9: Select a new batch $\mathcal{S}_{k+1} \subset \mathcal{B}$ with overlap ratio $o$
10: Compute the new curvature pair $(\boldsymbol{s}_k, \boldsymbol{t}_k)$ as in (4), (5)
11: Compute the new validation loss $v_k$
12: if number of stored $v_k == m_{val}$ then
13:     Remove the oldest $v_k$
14: end if
15: Store the newest validation loss $v_k$
16: if $m_k < m_{max}$ & $Q$ then
17:     $m_k \leftarrow \alpha m_k$
18: end if
19: if $q_k == m_k$ then
20:     if $m_k \leq m_{reset}$ then
21:         Delete all stored pairs and $q_k \leftarrow 0$
22:     else
23:         Remove the oldest pair and $q_k \leftarrow q_k - 1$
24:     end if
25: end if
26: Store the newest pair and $q_k \leftarrow q_k + 1$
27: $k \leftarrow k + 1$
```

a ReLU and a 2D max pooling layer. The function $C(\cdot)$ in (1) minimized during training is the cross-entropy loss function (Bishop (2006)), with $c(\cdot)$ defined as

$$c(l^{(i)}, \hat{l}^{(i)}) = -\{l^{(i)} \ln(\hat{l}^{(i)}) + (1 - l^{(i)}) \ln(1 - \hat{l}^{(i)})\}.$$

The algorithms were written in PyTorch. The data augmentation of the TRASH dataset was performed using Keras. The experiments with the CNN were executed on an nVidia P40-4Q virtual GPU, while the experiments with the MLP were executed on a local desktop with an Intel i7-6700 CPU and 16 GB of RAM. The code is available on GitHub [2].

Two metrics are used to assess algorithm performance: the correct classification rate (CCR) and the rank (RNK). CCR is defined as

$$\text{CCR} = \frac{100}{|\mathcal{T}|} \sum_{i \in \mathcal{T}} z^{(i)}, \quad z^{(i)} = \begin{cases} 1, & \text{if } l^{(i)} = \hat{l}^{(i)} \\ 0, & \text{otherwise} \end{cases}$$

where $\mathcal{T}$ is the test set. To evaluate RNK, for each simulation (60 in our experiments) we rank the five candidate

[1] https://github.com/hjmshi/PyTorch-LBFGS
[2] https://github.com/fedezocco/AdaMemLBFGS-PyTorch
[3] Downloaded from: https://scikit-learn.org/stable/datasets/
[4] Downloaded from: https://www.cs.ucr.edu/~eamonn/time_series_data/
[5] MNIST-R is a rescaled version of MNIST, which has 28 × 28 × 1 features.
[6] Downloaded from: https://keras.io/datasets/
[7] TRASH-RA is a rescaled and augmented version of TRASH, which has 512 × 384 × 3 features and 2527 images in total.
[8] Downloaded from: https://github.com/garythung/trashnet
[9] Kernel size = 5×5, kernels per layer = 1, stride = 1, dilatation = 1, padding = 0, pool size = 2×2, pool stride = 2.

Table 1. Overview of the case study datasets.

Dataset	# samples (train; test)	# features	# classes	Source
CANCER	(484; 85)	30	2	Street et al. (1993) [3]
ETCH	(1865; 329)	2046	3	Puggini and McLoone (2015)
WAFERS	(6164; 1000)	152	2	Olszewski (2001) [4]
MNIST-R [5]	(60000; 10000)	32 × 32 × 1	10	LeCun et al. (1998) [6]
CIFAR10	(50000; 10000)	32 × 32 × 3	10	Krizhevsky (2009) [6]
TRASH-RA [7]	(45107; 7960)	32 × 32 × 3	6	Yang and Thung (2016) [8]

Table 2. Description of the design of the six experiments considered, where parameters h, $\bar{m}$, n_{BFGS}, and n_{Adam} are the number of MLP hidden layer neurons, the maximum memory size for L-BFGS algorithms when m is constant (i.e. MB and MB-R), the number of iterations used for L-BFGS methods, and the number of iterations used for Adam, respectively.

	Dataset	Model	Hyperparameters
Experiment 1	CANCER	MLP	$\alpha = 2$, $m_0 = 1$, $m_{max} = 32$, $m_{val} = 5$, $m_{reset} = 8$ | $n_{BFGS} = 200$, $o = 0.45$, $\bar{m} = 10$, $r_{BFGS} = 256$, $\eta_{BFGS} = 0.5$ | $n_{Adam} = 200$, $r_{Adam} = 64$, $\eta_{Adam} = 0.02$ | $h = 35$
Experiment 2	ETCH	MLP	$\alpha = 2$, $m_0 = 1$, $m_{max} = 32$, $m_{val} = 5$, $m_{reset} = 8$ | $n_{BFGS} = 300$, $o = 0.45$, $\bar{m} = 10$, $r_{BFGS} = 512$, $\eta_{BFGS} = 0.5$ | $n_{Adam} = 300$, $r_{Adam} = 64$, $\eta_{Adam} = 0.03$ | $h = 320$
Experiment 3	WAFERS	MLP	$\alpha = 2$, $m_0 = 1$, $m_{max} = 32$, $m_{val} = 5$, $m_{reset} = 8$ | $n_{BFGS} = 60$, $o = 0.4$, $\bar{m} = 10$, $r_{BFGS} = 512$, $\eta_{BFGS} = 1$ | $n_{Adam} = 70$, $r_{Adam} = 64$, $\eta_{Adam} = 0.03$ | $h = 5$
Experiment 4	MNIST-R	CNN [9]	$\alpha = 2$, $m_0 = 1$, $m_{max} = 32$, $m_{val} = 5$, $m_{reset} = 8$ | $n_{BFGS} = 70$, $o = 0.25$, $\bar{m} = 10$, $r_{BFGS} = 8192$, $\eta_{BFGS} = 1$ | $n_{Adam} = 80$, $r_{Adam} = 128$, $\eta_{Adam} = 0.001$
Experiment 5	CIFAR10	CNN [9]	$\alpha = 2$, $m_0 = 1$, $m_{max} = 32$, $m_{val} = 5$, $m_{reset} = 8$ | $n_{BFGS} = 500$, $o = 0.25$, $\bar{m} = 10$, $r_{BFGS} = 8192$, $\eta_{BFGS} = 1$ | $n_{Adam} = 1000$, $r_{Adam} = 128$, $\eta_{Adam} = 0.001$
Experiment 6	TRASH-RA	CNN [9]	$\alpha = 2$, $m_0 = 1$, $m_{max} = 32$, $m_{val} = 5$, $m_{reset} = 8$ | $n_{BFGS} = 900$, $o = 0.25$, $\bar{m} = 10$, $r_{BFGS} = 8192$, $\eta_{BFGS} = 1$ | $n_{Adam} = 1200$, $r_{Adam} = 128$, $\eta_{Adam} = 0.001$

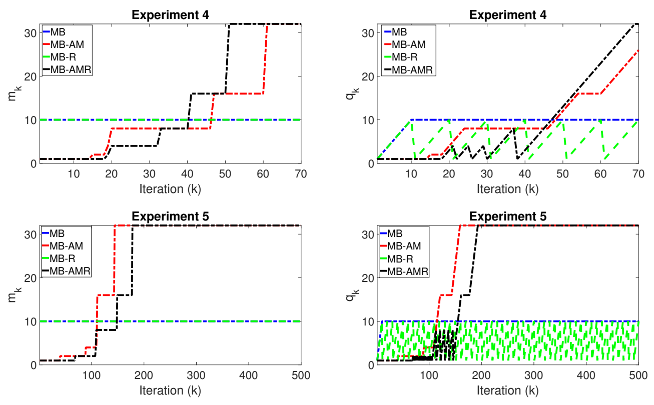

Fig. 1. Plot of the memory size m_k (left) and the number of stored curvature pairs q_k (right) as a function of k for selected experiments in Fig. 2.

methods in descending order such that the one with highest CCR gets RNK = 1, and the one with the lowest CCR gets RNK = 5. Thus, $1 \leq \text{RNK} \leq 5$.

4.2 Results

Fig. 1 shows m_k and the number of stored curvature pairs as a function of the number of iterations, k, while Fig. 2 shows the training loss, the test loss and the CCR as a function of k for a single training run of each experiment. Even though these figures do not permit statistically significant conclusions to be drawn because they depict a single simulation, they are useful in revealing the general behavior of the algorithms.

To evaluate the regularizing property of each method 60 Monte Carlo simulations were performed. Table 3 reports the mean and standard deviation of the final test CCR and RNK over these simulations, where "final" means the value once the training is completed. The last row of the table gives the mean and standard deviation over all six experiments, summarising the overall performance of each training algorithm.

The results vary considerably across the six experiments, with no single algorithm dominating. Adam achieves the best mean CCR performance in experiments 1, 3 and 6, MB-AM in experiments 4 and 5, and MB in experiment 2. Overall, MB-AM, MB-AMR and Adam yield similar mean CCR performance (74.1%-74.7%), with Adam having much lower variance in performance across the Monte Carlo simulations than the L-BFGS methods. MB-R is the worst performing training algorithm with a CCR of 69%, followed by MB at 71.4%. The fact that MB is more prone to converging to inferior solutions than the adaptive memory L-BFGS implementations is also reflected in the higher standard deviation in CCR (17.7% versus 11.9% for MB-AM, for example).

In terms of the rank metric, RNK, which considers the relative performance of each algorithm and is therefore less susceptible to outliers than CCR, MB-AM is the most consistently performing algorithm with a mean rank of 2.55, followed by MB-AMR with a mean rank of 2.58. MB is third at 2.8 and Adam is fourth at 3.2.

Table 4 shows the mean training computation time and, in parentheses, the mean time per iteration of the L-BFGS algorithms over the 60 Monte Carlo simulations normalized by the corresponding mean times for Adam. The algorithms have similar computation times for the three smaller dimension (MLP based) experiments, whereas MB-AM and MB-AMR are marginally faster than MB

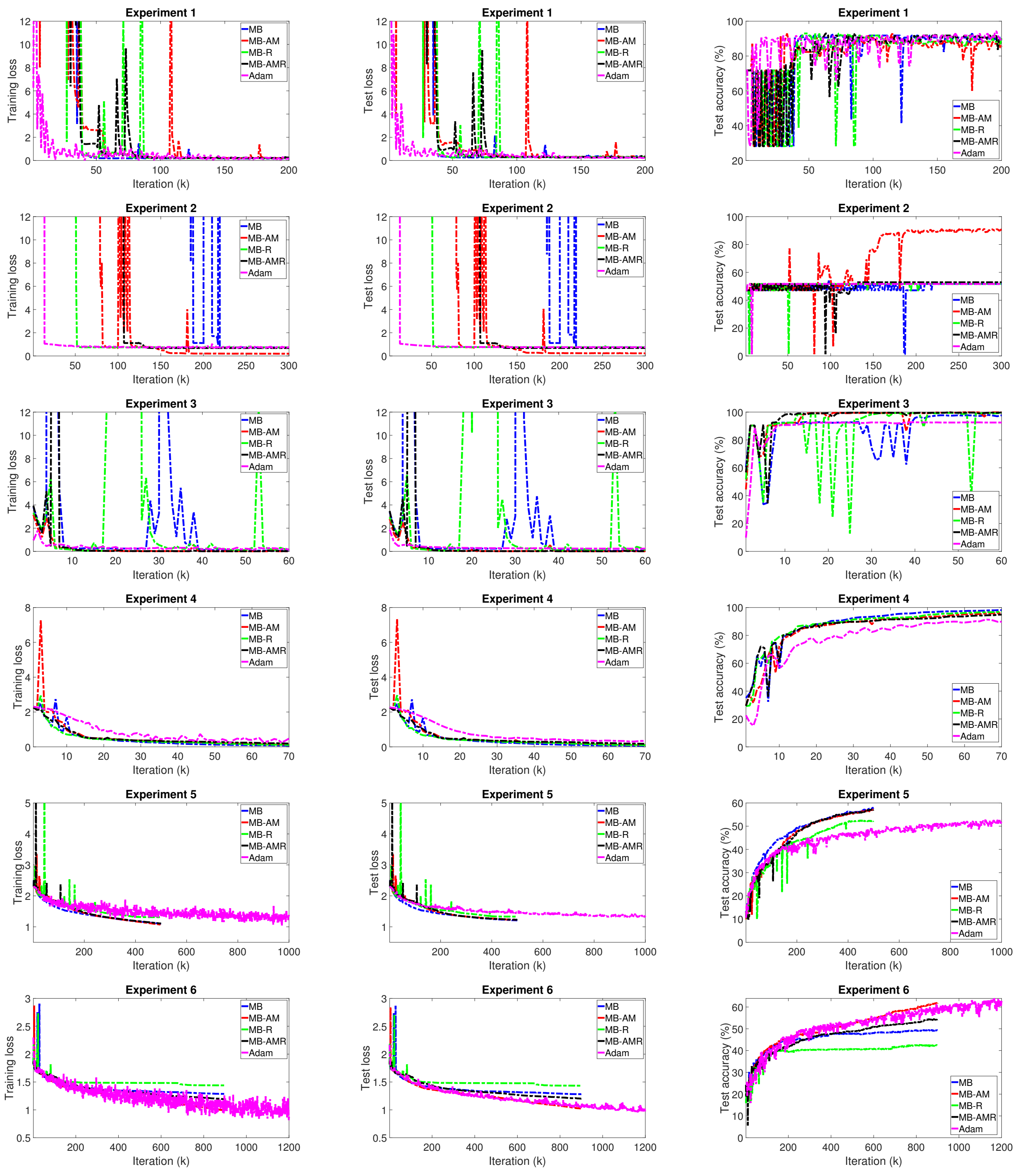

Fig. 2. The evolution of the training and test dataset cross-entropy loss and the test dataset CCR for a single simulation of each the six experiments considered in the study.

Table 3. Mean (standard deviation) of CCR and RNK over 60 Monte Carlo simulations. The best results for each experiment are highlighted in bold.

	MB		MB-AM		MB-R		MB-AMR		Adam	
	CCR	RNK	CCR	RNK	CCR	RNK	CCR	RNK	CCR	RNK
Exp. 1	80.9(15.8)	3.3(1.48)	85.0(13.8)	2.5(1.44)	84.7(13.4)	3.0(1.36)	85.8(12.3)	2.6(1.32)	**90.1(1.4)**	**1.9(1.23)**
Exp. 2	**65.0(21.3)**	**2.3(1.45)**	61.5(15.5)	2.6(1.14)	59.6(15.7)	2.4(1.17)	58.2(19.8)	2.7(1.31)	53.5(0.0)	4.1(0.87)
Exp. 3	87.5(21.6)	3.8(1.22)	91.5(19.3)	2.7(1.46)	95.9(9.8)	2.9(1.40)	95.0(12.5)	**2.2(1.43)**	**97.8(1.1)**	3.3(1.07)
Exp. 4	87.6(26.1)	**1.8(1.24)**	**92.4(15.4)**	2.8(1.14)	85.0(28.2)	2.7(1.23)	92.1(15.2)	3.1(1.05)	91.6(0.9)	4.6(0.72)
Exp. 5	57.9(9.2)	**1.9(1.10)**	**59.5(1.8)**	2.0(0.75)	48.9(9.3)	4.6(0.52)	57.8(6.8)	2.3(1.04)	53.1(1.2)	4.1(0.64)
Exp. 6	49.6(12.2)	3.4(1.03)	55.7(5.6)	2.7(0.89)	39.9(5.0)	4.9(0.37)	55.5(6.4)	2.6(0.89)	**62.4(1.6)**	**1.3(0.70)**
Mean	71.4(17.7)	2.8(1.25)	74.3(11.9)	**2.6(1.14)**	69.0(13.6)	3.4(1.01)	74.1(12.2)	**2.6(1.17)**	**74.7(1.0)**	3.2(0.87)

Table 4. Mean training times (and in parentheses, the mean time per iteration) for each L-BFGS algorithm normalized with respect to the corresponding values for Adam.

	MB	MB-AM	MB-R	MB-AMR
Exp. 1	2.0(2.0)	2.1(2.1)	**1.9**(1.9)	2.1(2.1)
Exp. 2	5.3(5.3)	5.8(5.8)	**5.1**(5.1)	5.8(5.8)
Exp. 3	2.7(3.1)	**2.6**(**3.0**)	2.7(3.1)	**2.6**(**3.0**)
Exp. 4	23.2(26.5)	22.3(25.5)	23.0(26.2)	**22.2**(**25.4**)
Exp. 5	15.6(31.2)	**14.8**(**29.7**)	15.6(31.2)	14.9(29.8)
Exp. 6	22.0(29.4)	**21.2**(**28.3**)	22.0(29.3)	21.3(28.4)

when training the larger CNN based problems (3%-4%). This is a consequence of the low memory usage in early iterations which speeds up the L-BFGS two-loop recursion computation. The use of large batches and the computation of the approximated second order information make the L-BFGS methods substantially more computationally intensive than Adam and the advantage of the latter increases for the larger experiments. For example, Adam is more than 22 times faster than the L-BFGS algorithms for the MNIST case study (Experiment 4).

5. CONCLUSIONS

This paper has proposed three variants of the multi-batch L-BFGS algorithm of Berahas et al. (2016) based on the idea of progressive use of curvature information through a *dev-increase* scheme, and periodic memory resetting as introduced in McLoone and Irwin (1999). The experimental results show that the *dev-increase* adaptive memory scheme improves the generalisation performance and consistency of results obtained with L-BFGS and also slightly reduces its computational complexity. Periodic memory resetting is shown to be detrimental to performance, with MB-AMR marginally inferior to MB-AM and MB-R substantially inferior to MB. The overall ordering of algorithms in terms of generalisation performance is MB-AM, MB-AMR, MB, Adam and MB-R. In contrast, in terms of computational efficiency the order is Adam, MB-AM, MB-AMR, MB-R, and MB, with Adam vastly superior to the other training algorithms. Therefore, Adam remains the algorithm of choice for training large neural networks, and only when the resulting accuracy is not satisfactory, should the use of MB-AM be considered.

MB-AM has the drawback of requiring three additional hyperparameters compared to MB (α, m_0/m_{max} and m_{val}). The values used in this study are empirically defined and appropriate for a first investigation of the method. Future research will look at the automated selection of these parameters, and the potential for integrating MB-AM with progressive batching (Bollapragada et al. (2018)).

REFERENCES

Agarwal, A., Chapelle, O., Dudík, M., and Langford, J. (2014). A reliable effective terascale linear learning system. *The Journal of Machine Learning Research*, 15(1), 1111–1133.

Berahas, A.S., Nocedal, J., and Takác, M. (2016). A multi-batch L-BFGS method for machine learning. In *Advances in Neural Information Processing Systems*, 1055–1063.

Bishop, C.M. (2006). *Pattern recognition and machine learning*. Springer.

Bollapragada, R., Mudigere, D., Nocedal, J., Shi, H.J.M., and Tang, P.T.P. (2018). A progressive batching L-BFGS method for machine learning. In *35th International Conference on Machine Learning, ICML 2018*, 989–1013.

Dodge, S. and Karam, L. (2017). A study and comparison of human and deep learning recognition performance under visual distortions. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, 1–7. IEEE.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

Kingma, D.P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical report, University of Toronto.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278–2324.

Liu, D.C. and Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1-3), 503–528.

McLoone, S., Brown, M.D., Irwin, G., and Lightbody, A. (1998). A hybrid linear/nonlinear training algorithm for feedforward neural networks. *IEEE Transactions on Neural Networks*, 9(4), 669–684.

McLoone, S. and Irwin, G. (1999). A variable memory quasi-Newton training algorithm. *Neural Processing Letters*, 9(1), 77–89.

Nocedal, J. and Wright, S. (2006). *Numerical optimization*. Springer Science & Business Media.

Olszewski, R.T. (2001). *Generalized feature extraction for structural pattern recognition in time-series data*. PhD. thesis, Carnegie Mellon University.

Puggini, L. and McLoone, S. (2015). Extreme learning machines for virtual metrology and etch rate prediction. In *2015 26th Irish Signals and Systems Conference (ISSC)*, 1–6.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

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

Street, W.N., Wolberg, W.H., and Mangasarian, O.L. (1993). Nuclear feature extraction for breast tumor diagnosis. In *Biomedical Image Processing and Biomedical Visualization*, volume 1905, 861–870.

Wilson, A.C., Roelofs, R., Stern, M., Srebro, N., and Recht, B. (2017). The marginal value of adaptive gradient methods in machine learning. In *Advances in Neural Information Processing Systems*, 4148–4158.

Yang, M. and Thung, G. (2016). Classification of trash for recyclability status. Technical report, Stanford University.

Yousefian, F., Nedić, A., and Shanbhag, U. (2017). On stochastic and deterministic quasi-Newton methods for non-strongly convex optimization: convergence and rate analysis. *arXiv preprint arXiv:1710.05509*.