SMALL STEPS AND GIANT LEAPS: MINIMAL NEWTON SOLVERS FOR DEEP LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

We propose a fast second-order method that can be used as a drop-in replacement for current deep learning solvers. Compared to stochastic gradient descent (SGD), it only requires two additional forward-mode automatic differentiation operations per iteration, which has a computational cost comparable to two standard forward passes and is easy to implement. Our method addresses long-standing issues with current second-order solvers, which invert an approximate Hessian matrix every iteration exactly or by conjugate-gradient methods, procedures that are much slower than a SGD step. Instead, we propose to keep a single estimate of the gradient projected by the inverse Hessian matrix, and update it once per iteration with just two passes over the network. This estimate has the same size and is similar to the momentum variable that is commonly used in SGD. No estimate of the Hessian is maintained. We first validate our method, called CURVEBALL, on small problems with known solutions (noisy Rosenbrock function and degenerate 2-layer linear networks), where current deep learning solvers struggle. We then train several large models on CIFAR and ImageNet, including ResNet and VGG-f networks, where we demonstrate faster convergence with no hyperparameter tuning. We also show our optimiser's generality by testing on a large set of randomly-generated architectures.

1 INTRODUCTION

Stochastic Gradient Descent (SGD) and back-propagation (LeCun et al., 1998) are the algorithmic backbone of current deep network training. The success of deep learning demonstrates the power of this combination, which has been successfully applied on various tasks with large datasets and very deep networks (He et al., 2016).

Yet, while SGD has many advantages, speed of convergence (in terms of number of iterations) is not necessarily one of them. While individual SGD iterations are very quick to compute and lead to rapid progress at the beginning of the optimisation, soon the latter reaches a slower phase where further improvements are achieved slowly. This can be attributed to entering regions of the parameter space where the objective function is poorly scaled. In such cases, rapid progress would require vastly different step sizes for different directions in parameter space, which SGD cannot deliver.

Second-order methods, such as Newton's method and its variants, eliminate this issue by rescaling the gradient according to the local curvature of the objective function. For a scalar loss in $\mathbb{R}$, this rescaling takes the form $H^{-1}J$ where H is the Hessian matrix (second-order derivatives) or an approximation of the local curvature in the objective space, and J is the gradient of the objective. They can in fact achieve local scale-invariance, and make provably better progress in the regions where gradient descent stalls (Wright & Nocedal, 1999). While they are unmatched in other domains, there are several obstacles to their application to deep models. First, it is impractical to invert or even store the Hessian, since it grows quadratically with the number of parameters, and there are typically millions of them. Second, any Hessian estimate is necessarily noisy and ill-conditioned due to stochastic sampling, to which classic inversion methods such as conjugate-gradient are not robust.

In this paper, we propose a new algorithm that can overcome these difficulties and make second order optimisation practical for deep learning. We show in particular how to avoid the storage of any estimate of the Hessian matrix or its inverse. Instead, we treat the computation of the Newton update, $H^{-1}J$, as solving a linear system that itself can be solved via gradient descent. The cost of solving this system is amortized over time by interleaving its steps with the parameter update steps. Our

proposed method adds little overhead, since a Hessian-vector product can be implemented for modern networks with just two steps of automatic differentiation. Interestingly, we show that our method is equivalent to momentum SGD (also known as the heavy-ball method) with a single additional term, accounting for curvature. For this reason we named our method CURVEBALL. Unlike other proposals, the total memory footprint is as small as that of momentum SGD.

This paper is structured as follows. We introduce relevant technical background in sec. 2, and present our method in sec. 3. We evaluate our method and show experimental results in sec. 4. Related work is discussed in sec. 5. Finally we summarise our findings in sec. 6.

2 BACKGROUND

In order to make the description of our method self-contained, we succinctly summarise a few standard concepts in optimisation. Our goal is to find the optimal parameters of a model (*e.g.* a neural network) $\phi : \mathbb{R}^p \to \mathbb{R}^o$, with p parameters $w \in \mathbb{R}^p$ and o outputs (the notation does not show the dependency on the training data, which is subsumed in ϕ for compactness). The quality of the outputs is evaluated by a loss function $L : \mathbb{R}^o \to \mathbb{R}$, so finding w is reduced to the optimisation problem: [1]

$$w^* = \arg\min_w L(\phi(w)) = \arg\min_w f(w). \tag{1}$$

Perhaps the simplest algorithm to optimise eq. 1 is gradient descent (GD). GD updates the parameters using the iteration $w \leftarrow w - \beta J(w)$, where $\beta > 0$ is the learning rate and $J(w) \in \mathbb{R}^p$ is the gradient (or Jacobian) of the objective function f with respect to the parameters w. A useful variant is to augment GD with a *momentum* variable z, which can be interpreted as a decaying average of past gradients:[2]

$$z \leftarrow \rho z - \beta J(w) \tag{2}$$

$$w \leftarrow w + z \tag{3}$$

Momentum GD, as given by eq. 2-3, can be shown to have faster convergence than GD, remaining stable under higher learning rates, and exhibits somewhat better resistance to poor scaling of the objective function (Goh, 2017). One important aspect is that these advantages cost almost no additional computation and only a modest additional memory, which explains why it is widely used in practice.

In neural networks, GD is usually replaced by its stochastic version (SGD), where at each iteration one computes the gradient not of the model $f = L(\phi(w))$, but of the model $f_t = L_t(\phi_t(w))$ assessed on a small batch of samples, drawn at random from the training set.

2.1 SECOND-ORDER OPTIMISATION

As mentioned in section 1, the Newton method is similar to GD, but steers the gradient by the inverse Hessian matrix, computing $H^{-1}J$ as a descent direction. However, inverting the Hessian may be numerically unstable or the inverse may not even exist. To address this issue, the Hessian is usually regularized, obtaining what is known as the *Levenberg* (Moré, 1978) method:

$$z = -(H + \lambda I)^{-1} J, \tag{4}$$

$$w \leftarrow w + z, \tag{5}$$

where $H \in \mathbb{R}^{p \times p}$, $J \in \mathbb{R}^p$, and $I \in \mathbb{R}^{p \times p}$ is the identity matrix. Note that, unlike for momentum GD (eq. 2), the new step z is independent of the previous step. To avoid burdensome notation, we omit the w argument in $H(w)$ and $J(w)$, but they must be recomputed at each iteration. Intuitively, the effect of eq. 4 is to rescale the step appropriately for different directions — directions with high curvature require small steps, while directions with low curvature require large steps to make progress.

Note also that Levenberg's regularization loses the scale-invariance of the original Newton method, meaning that rescaling the function f changes the scale of the gradient and hence the regularised descent direction chosen by the method. An alternative that alleviates this issue is *Levenberg-Marquardt*, which replaces I in eq. 4 with $\mathrm{diag}(H)$. For non-convex functions such as deep networks, these methods only converge to a local minimum when the Hessian is positive-semidefinite (PSD).

[1] We omit the optional regulariser term for brevity, but this does not materially change our derivations.

[2] Some sources use an alternative form with β in eq. 3 instead (equivalent by a re-parametrisation of z).

2.2 AUTOMATIC DIFFERENTIATION AND BACK-PROPAGATION

In order to introduce fast computations involving the Hessian, we must take a short digression into how Jacobians are computed. The Jacobian of $L(\phi(w))$ (eq. 1) is generally computed as $J = J_\phi J_L$ where $J_\phi \in \mathbb{R}^{p \times o}$ and $J_L \in \mathbb{R}^{o \times 1}$ are the Jacobians of the model and loss, respectively. In practice, a Jacobian is never formed explicitly, but Jacobian-vector products Jv are implemented with the back-propagation algorithm. We define

$$\overleftarrow{\mathrm{AD}}(v) = Jv \tag{6}$$

as the *reverse-mode automatic differentiation* (RMAD) operation, commonly known as *back-propagation*. A perhaps lesser known alternative is *forward-mode automatic differentiation* (FMAD), which computes a vector-Jacobian product, from the other direction:

$$\overrightarrow{\mathrm{AD}}(v') = v'J \tag{7}$$

This variant is less commonly-known in deep learning as RMAD is appropriate to compute the derivatives of a scalar-valued function, such as the learning objective, whereas FMAD is more appropriate for vector-valued functions of a scalar argument. However, we will show later that FMAD is relevant in calculations involving the Hessian.

The only difference between RMAD and FMAD is the direction of associativity of the multiplication: FMAD propagates gradients in the forward direction, while RMAD (or back-propagation) does it in the backward direction. For example, for the composition of functions $a \circ b \circ c$,

$$\begin{aligned} \overrightarrow{\mathrm{AD}}_{a \circ b \circ c}(v) &= \left(\left(v J_a\right) J_b\right) J_c \\ \overleftarrow{\mathrm{AD}}_{a \circ b \circ c}(v') &= J_a \left(J_b \left(J_c v'\right)\right) \end{aligned}$$

Because of this, both operations have similar computational overhead, and can be implemented similarly. Note that, because the loss is a scalar function, the starting projection vector v in back-propagation is a scalar and we set $v = 1$. For intermediate computations, however, it is generally a (vectorized) tensor of gradients.

2.3 FAST HESSIAN-VECTOR PRODUCTS

Since the Hessian of learning objectives involving deep networks is not necessarily positive semi-definite (PSD), it is common to use the Gauss-Newton approximation (Wright & Nocedal, 1999; Martens & Grosse, 2015; Botev et al., 2017):

$$\hat{H} = J_\phi H_L J_\phi^T, \tag{8}$$

When H_L is PSD, which is the case for all convex losses (*e.g.* logistic loss, L^p distance), the resulting $\hat{H}$ is PSD by construction. Even though it is approximate, it is still desirable to use eq. 8 since it is guaranteed to be PSD and thus prevents second-order methods from being attracted to saddle-points (Dauphin et al., 2014).

For the method that we propose, and indeed for any method that implicitly inverts the Hessian (or its Gauss-Newton approximation), only computing Hessian-vector products $\hat{H}v$ is required. As such, eq. 8 takes a very convenient form:

$$\begin{aligned} \hat{H}v &= J_\phi \left(H_L \left(J_\phi^T v\right)\right) && (9) \\ &= \overleftarrow{\mathrm{AD}}_\phi \left(H_L \left(\overrightarrow{\mathrm{AD}}_\phi(v)\right)\right). && (10) \end{aligned}$$

The cost of eq. 10 is thus equivalent to that of two back-propagation operations. This is similar to a classic result (Pearlmutter, 1994; Schraudolph, 2002), but written in terms of common automatic differentiation operations. The intermediate matrix-vector product $H_L u$ has negligible cost: for example, for the squared-distance loss, $H_L = 2I \Rightarrow H_L u = 2u$. Similarly, for the multinomial logistic loss we have $H_L = \mathrm{diag}(p) - pp^T \Rightarrow H_L u = p \odot u - p(p^T u)$, where p is the vector of predictions from a softmax layer and $\odot$ is the element-wise product. These products thus require only element-wise operations.

Algorithm 1: CURVEBALL (proposed).

1: $z_0 = \mathbf{0}$
2: **for** $t = 0, ..., T-1$ **do**
3: $\quad \Delta_z = H(w_t)z_t + J(w_t)$
4: $\quad z_{t+1} = \rho z_t - \beta \Delta_z$
5: $\quad w_{t+1} = w_t + z_{t+1}$
6: **end for**

Algorithm 2: Simplified Hessian-free method.

1: **for** $t = 0, ..., T-1$ **do**
2: $\quad z_0 = -J(w_t)$
3: $\quad$ **for** $r = 0, ..., R-1$ (or convergence) **do**
4: $\quad\quad z_{r+1} = \mathrm{CG}(z_r, H(w_t)z_r, J(w_t))$
5: $\quad$ **end for**
6: $\quad w_{t+1} = w_t + z_R$
7: **end for**

3 METHOD

This section presents our main contribution: a method that minimizes a second-order Taylor expansion of the objective (like the Newton variants from section 2.1), but at a much reduced computational and memory cost, suitable for very large-scale problems. The result of taking a step z away from a starting point w can be modelled with a second-order Taylor expansion of the objective f:

$$f(w+z) \simeq \hat{f}(w,\, z) = f(w) + z^T J(w) + \tfrac{1}{2} z^T H(w) z \tag{11}$$

Most second-order methods seek the update z that minimizes $\hat{f}$, by ignoring the higher-order terms:

$$z = \arg\min_{z'} \hat{f}(z') = \arg\min_{z'} \tfrac{1}{2} z'^T H z' + z'^T J \tag{12}$$

In general, a step z is found by minimizing eq. 12, either via explicit inversion $H^{-1}J$ (Martens & Grosse, 2015; Botev et al., 2017) or the conjugate-gradient (CG) method (Martens, 2010). The later approach, called the Hessian-free method (also Truncated Newton or Newton-CG (Wright & Nocedal, 1999)) is the most economical in terms of memory, since it only needs access to Hessian-vector products (section 2.3). A high-level view is illustrated in Algorithm 2, where CG stands for one step of conjugate-gradient (a stopping condition, line search and some intermediate variables were omitted for clarity). Notice that for every update of w (outer loop), Algorithm 2 must perform several steps of CG (inner loop) to find a *single* search direction z. If CG takes N steps on average, then Algorithm 2 will be slower than SGD by a factor of at least N, which can easily be an order of magnitude.

We propose a number of changes in order to eliminate this costly inner loop. The first is to reuse the previous search direction z to warm-start the inner iterations, instead of resetting z each time (Algorithm 2, line 2). If z does not change abruptly, then this should help reduce the number of CG iterations, by starting closer to the solution. The second change is to use this fact to dramatically reduce the inner loop iterations to just one ($R = 1$). A different interpretation is that we now *interleave* the updates of the search direction z and parameters w (Algorithm 1, lines 4 and 5), instead of nesting them (Algorithm 2, lines 4 and 6).

Unfortunately, this change loses the guarantees of the CG method, which depend on the starting point being $z_0 = -J(w_t)$ (Wright & Nocedal, 1999). This loss of guarantee was verified in practice, as we found the resulting algorithm extremely unstable. Our third change is then to replace CG with gradient descent, which has no such dependency. Differentiating eq. 12 w.r.t. z yields:

$$\triangle_z = J_{\hat{f}(z)} = Hz + J \tag{13}$$

Applying these changes to the Hessian-free method (Algorithm 2) results in Algorithm 1. By contrasting it to momentum GD (eq. 2-3), we can see that it is equivalent, except for an extra curvature term $H(w)z$. In order to establish this equivalence, we introduced a factor ρ that decays z each step (Algorithm 1, line 4), whereas a pure gradient descent step on z would not include this factor. We can obtain it formally by simply regularizing the quadratic model in eq. 11 with the term $(1-\rho)\,\|z\|^2$, which is small when $\rho \simeq 1$ (the recommended setting for the momentum parameter (He et al., 2016)). Due to the addition of curvature to momentum GD, which is also known as the heavy-ball method, we name our algorithm CURVEBALL.

Implementation. Using the fast Hessian-vector products from section 2.3, it is easy to implement eq. 13, including a regularization term λI (section 2.1). We can further improve eq. 13 by grouping

Table 1: **Optimiser comparison on small degenerate datasets.** For each optimiser, we report the mean$\pm$standard deviation of the number of iterates taken to reach the solution. For the stochastic Rosenbrock function, $\mathcal{U}[\lambda_1, \lambda_2]$ denotes noise drawn from $\mathcal{U}[\lambda_1, \lambda_2]$ (see Sec. 4 for details)

	Deterministic	Rosenbrock $\mathcal{U}[0,1]$	$\mathcal{U}[0,3]$	Raihimi & Recht
SGD + momentum	370 ± 40	846 ± 217	4069 ± 565	95 ± 2
Adam Kingma & Ba (2014)	799 ± 160.5	1290 ± 476	2750 ± 257	95 ± 5
Levenberg-Marquardt Moré (1978)	16 ± 4	$\mathbf{8 \pm 3}$	$\mathbf{11 \pm 5}$	$\mathbf{9 \pm 4}$
BFGS Wright & Nocedal (1999)	19 ± 4	44 ± 21	63 ± 29	43 ± 21
CURVEBALL (proposed)	$\mathbf{13 \pm 0.5}$	12 ± 1	13 ± 1	35 ± 11

the operations to minimize the number of automatic differentiation (back-propagation) steps:

$$\triangle_z = \left(J_\phi H_L J_\phi^T + \lambda I\right) z + J_\phi J_L \tag{14}$$

$$= J_\phi \left(H_L J_\phi^T z + J_L\right) + \lambda z \tag{15}$$

In this way, the total number of passes over the model is two: we compute $J_\phi v$ and $J_\phi^T v'$ products, implemented respectively as one RMAD (back-propagation) and one FMAD operation (section 2.2).

Automatic ρ and β hyper-parameters in closed form. Our proposed method introduces a few hyper-parameters, which just like with SGD, would require tuning for different settings. Ideally, we would like to have no tuning at all. Fortunately, the quadratic minimization interpretation in eq. 12 allows us to draw on standard results in optimisation (Wright & Nocedal, 1999). At any given step, the optimal ρ and β can be obtained by solving a 2×2 linear system Martens & Grosse (2015):

$$\begin{bmatrix} \beta \\ -\rho \end{bmatrix} = \begin{bmatrix} \Delta_z^T \hat{H} \Delta_z & z^T \hat{H} \Delta_z \\ z^T \hat{H} \Delta_z & z^T \hat{H} z \end{bmatrix}^{-1} \begin{bmatrix} J^T \Delta_z \\ J^T z \end{bmatrix} \tag{16}$$

Note that, in calculating the proposed update (eq. 14), the quantities Δ_z, $J_\phi^T z$ and J_L have already been computed and can now be reused. Together with the fact that $\hat{H} = J_\phi H_L J_\phi^T$, this means that the elements of the above 2×2 matrix can be computed with only one additional forward pass.

Automatic λ hyper-parameter rescaling. The regularization term λI (eq. 4) can be interpreted as a trust-region (Wright & Nocedal, 1999). When the second-order approximation holds well, λ can be small, corresponding to an unregularized Hessian and a large trust-region. Conversely, a poor fit requires a correspondingly large λ. We can measure the difference (or ratio) between the objective change predicted by the quadratic fit ($\hat{f}$) and the real objective change (f), by computing $\gamma = \left(f(w+z) - f(w)\right) / \hat{f}(z)$. This requires one additional evaluation of the objective for $f(w+z)$, but otherwise relies only on previously computed quantities. This makes it a very attractive estimate of the trust region, with $\gamma = 1$ corresponding to a perfect approximation. Following Wright & Nocedal (1999), we evaluate γ every 5 iterations, decreasing λ by a factor of 0.999 when $\gamma > 3/2$, and increasing by the inverse factor when $\gamma < 1/2$. We noted that our algorithm is not very sensitive to the initial λ. In experiments using batch-normalization (section 4), we simply initialize it to one, otherwise setting it to 10.

Convergence. In addition to the usual absence of strong guarantees for non-convex problems, which applies in our setting (deep neural networks), there is an added difficulty due to the recursive nature of our algorithm (the interleaved w and z steps). Our method is a variant of the heavy-ball method (by adding a curvature term), which until very recently had resisted establishing global convergence rates that improve on gradient descent without momentum (Loizou & Richtárik (2017), table 1), and even then only for strongly convex or quadratic functions.

For this reason, we present proofs for two more tractable cases (Appendix A). The first is the global linear convergence of our method for convex quadratic functions, which allows a direct inspection of the region of convergence as well as its rate (Theorem A.1). The second establishes that, for convex functions, CURVEBALL's steps are always in a descent direction, when using the automatic hyper-parameter tuning of eq. 16 (Theorem A.2). We note that in practice, due to the Gauss-Newton approximation and the trust region (eq. 4), the effective Hessian is guaranteed to be positive-definite.

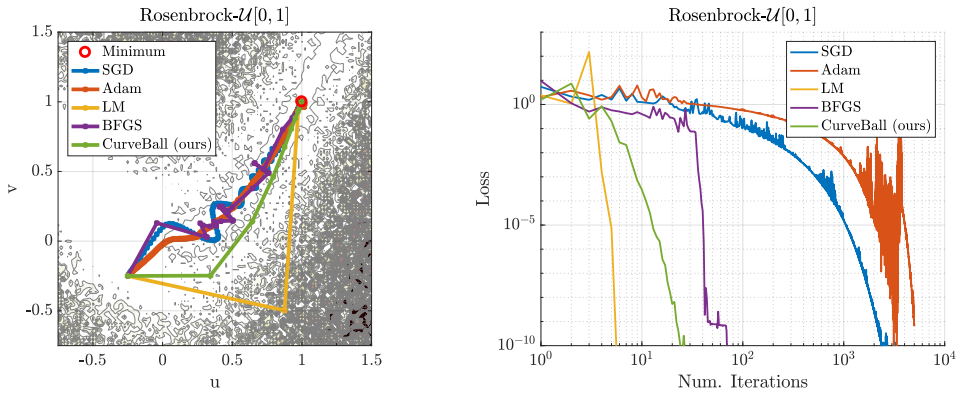

Figure 1: **Degenerate optimisation problems with known solutions**. Left: Trajectories on the Stochastic Rosenbrock function for different solvers (darker shaded regions denote higher function values). Right: evolution of the loss per iterations for the plotted trajectories.

Similarly to momentum SGD, our main claim as to the method's suitability for non-convex deep network optimisation is necessarily empirical, based on the extensive experiments in section 4, which show strong performance on several large-scale problems with no hyper-parameter tuning.

4 Experiments

Degenerate problems with known solutions. While the main purpose of our optimizer is its application to large-scale deep learning architecture, we begin by applying our method to problems of limited complexity, with the goal of exploring the strengths and weaknesses of our approach in an interpretable domain. We perform a comparison with two popular first order solvers — SGD with momentum and Adam Kingma & Ba (2014)[3], as well as with more traditional methods such as Levenberg-Marquardt and BFGS (Wright & Nocedal, 1999) (with cubic line-search). The first problem we consider is the search for the minimum of the two-dimensional Rosenbrock test function, which has the useful benefit of enabling us to visualise the trajectories found by each optimiser. Specifically, we use the stochastic variant of this function Yang & Deb (2010), $\mathcal{R} : \mathbb{R}^2 \to \mathbb{R}$:

$$\mathcal{R}(u, v) = (1 - u)^2 + 100\epsilon_i(v - u^2)^2, \tag{17}$$

where at each evaluation of the function, a noise sample ϵ_i is drawn from a uniform distribution $\mathcal{U}[\lambda_1, \lambda_2]$ with $\lambda_1, \lambda_2 \in \mathbb{R}$ (we can recover the deterministic Rosenbrock function with $\lambda_1 = \lambda_2 = 1$). To assess robustness to noise, we compare each optimiser on the deterministic formulation and two stochastic variants (with differing noise regimes). We also consider a second problem of interest, recently introduced by Rahimi & Recht (2017). It consists of fitting a deep network with only two linear layers to a dataset where sample inputs x are related to sample outputs y by the relation $y = Ax$, where A is an ill-conditioned matrix (with condition number $\epsilon = 10^5$).

The results are shown in Table 1. We use a grid-search to determine the best hyper-parameters for both SGD and Adam (reported in appendix B.1). We report the number of iterates taken to reach the solution, with a tolerance of $\tau = 10^{-4}$. Statistics are computed over 100 runs of each optimiser. We observe that first-order methods perform poorly in all cases, and moreover show a very high variance of results. The Levenberg-Marquardt (LM) optimiser generally performs best, although it is impractical for larger-scale problems. Our method delivers comparable performance despite avoiding a costly Hessian inversion. On the other hand, the performance of BFGS, which approximates the Hessian with a buffer of parameter updates, seems to correlate negatively with the level of noise.

Fig. 1 shows example trajectories. The slow, oscillating behaviour of first-order methods is noticeable, as well as the impact of noise on the BFGS steps. On the other hand, both our method and LM converge in few iterations.

[3]We also experimented with RMSProp (Tieleman & Hinton, 2012), AdaGrad (Duchi et al., 2011) and AdaDelta (Zeiler, 2012), but found that these methods consistently underperformed Adam and SGD on these "toy" problems.

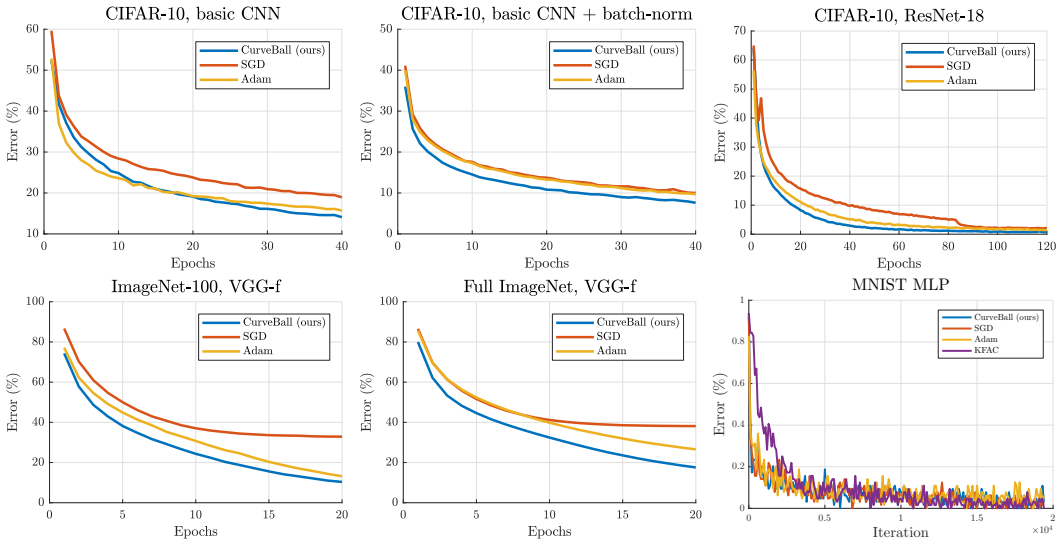

Figure 2: **Comparison with the different optimisers for various datasets and networks**. The evolution of the training error is shown, as it is the quantity being optimised. CURVEBALL performs well under a variety of realistic settings, including large-scale datasets (ImageNet), the presence of batch-normalization, and severely over-parameterised models (ResNet).

CIFAR. We now turn to the task of training deep-networks on more realistic datasets. Second-order methods are typically not used in such scenarios, due to the large number of parameters and stochastic sampling. We start with a basic 5-layer convolutional neural network (CNN).[4] We train this network for 20 epochs on CIFAR-10, with and without batch-normalization (which is known to improve optimisation) using for every experiment a mini-batch size of 128. To assess optimiser performance on larger models, we also train a much larger ResNet-18 model (He et al., 2016). As baselines, we picked SGD (with momentum) and Adam, which we found to outperform the competing first-order optimisers. Their learning rates are chosen from the set $10^{-k},\ k \in \mathbb{N}$ with a grid search for the basic CNN, while for the ResNet SGD uses the schedule recommended by the authors He et al. (2016). We focus on the training error, since it is the quantity being optimised by eq. 1 (validation error is discussed below). The results can be seen in Fig. 2 (top row). We observe that in each setting, CURVEBALL outperforms its competitors, in a manner that is robust to normalisation and model type.

ImageNet. To assess the practicality of our method at larger scales, we apply it to the classification task on the large-scale ImageNet dataset. We report results of training on both a medium-scale setting using a subset formed from the images of 100 randomly sampled classes as well as the large-scale setting, by training on the full dataset. Both experiments use the VGG-f architecture with mini-batch size of 128 and follow the settings described in Chatfield et al. (2014). The results are depicted in Fig. 2. We see that our method provides compelling performance against popular first order solvers in both cases, and that interestingly, its margin of improvement grows with the scale of the dataset.

Comparison to other second-order methods on MNIST. In order to compare ours with existing second-order methods, we use the public KFAC (Martens & Grosse, 2015) implementation made available by the authors and run a simple experiment on the MNIST dataset. In this scenario a four layer MLP (with output sizes 128-64-32-10) with hyperbolic tangent activations is trained on this classification task. We closely follow the same protocol as Martens & Grosse (2015) for layer initialisation and data normalisation, with batch size 64. We show results in Fig 2 (bottom row, right) with the best learning rate for each method. On this problem our method performs comparably to first order solvers, while KFAC makes less progress until it has stabilised its Fisher matrix estimation.

[4]The basic CNN has 5×5 filters and ReLU activations, and 3×3 max-pooling layers (with stride 2) after each of the first 3 convolutions. The number of output channels are, respectively, $(32, 32, 64, 64, 10)$.

Table 2: Best error in percentage (training/validation) for different models and optimisation methods. CURVEBALL λ denotes use of λ rescaling (sec. 3). Numbers in bracket show validation error with additional dropout regularisation (rate 0.3). The first three columns are trained on CIFAR-10, while the fourth is trained on ImageNet-100.

Model	Basic	Basic + BN	ResNet-18	VGG-f
CURVEBALL λ	**14.1** / 19.9	**7.6** / 16.3	**0.7** / 15.3 (13.5)	**10.3 / 33.5**
CURVEBALL	15.3 / **19.3**	9.4 / **15.8**	1.3 / 16.1	12.7 / 33.8
SGD	18.9 / 21.1	10.0 / 16.1	2.1 / **12.8**	32.9 / 41.7
Adam	15.7 / 19.7	9.6 / 16.1	1.4 / 14.0	13.2 / 35.9

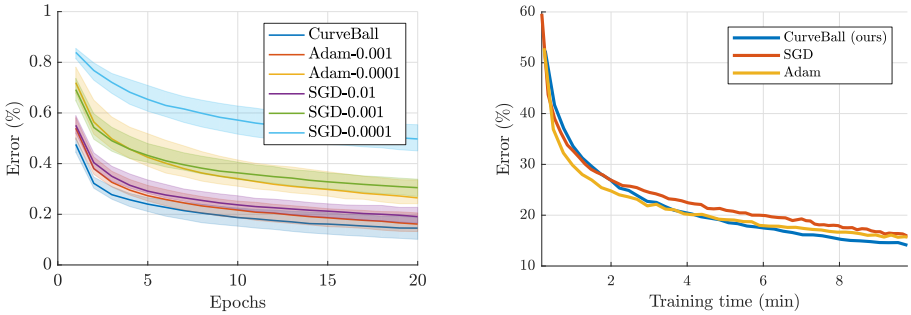

Figure 3: (Left) **Results of 50 randomly-generated architectures on CIFAR10.** The median (thick lines) and 25th-75th percentiles (shaded region) are shown. Numbers in the legend represent learning rates (fixed for CURVEBALL). (Right) **Training error vs. wall clock time** (basic CIFAR-10 model).

Random architecture results. It can be argued that standard architectures are biased to favour SGD, since it was used in the architecture searches, and architectures in which it failed to optimise were discarded He et al. (2016). It would be useful to assess the optimisers' ability to generalise across architectures, testing how well they perform regardless of the network model. We make an attempt in this direction by comparing the optimisers on 50 deep CNN architectures that are generated randomly (see appendix B.2 for details). In addition to being more architecture-agnostic, this makes any hand-tuning of hyper-parameters infeasible, which we believe to be a fair requirement for a dependable optimiser. The results on CIFAR10 are shown in figure 3 (left), as the median across all runs (thick lines) and 25th-75th percentiles (shaded regions). CURVEBALL consistently outperforms first-order methods, with the bulk of the achieved errors below those of SGD and Adam.

Wall-clock time. To provide an estimate of the relative efficiency of each model, Fig. 3 shows wall clock time on the basic CIFAR-10 model (without batch norm). Importantly, from a practical perspective, we observe that our method is competitive with first order solvers, while not requiring any tuning. Moreover, our prototype implementation includes custom FMAD operations which have not received the same degree of optimisation as RMAD (back-propagation), and could further benefit from careful engineering. We also experimented with a Hessian-free optimiser (based on conjugate gradients) Martens (2010). We show a comparison in logarithmic time in the appendix (Fig. 5). Due to the costly CG operation, which requires several passes through the network, it is an order of magnitude slower than the first-order methods and our own second-order method. This validates our initial motivation of designing a Hessian-free method without the inner CG loop (Section 3).

Overfitting and validation error While the focus of this work is optimisation, it is also of interest to compare the validation errors attained by the trained models – these are reported Table 2. We observe that models trained with the proposed method exhibit better training and validation error on most models, with the exception of ResNet where overfitting plays a more significant role. However, we note that this could be addressed with better regularisation, and we show one such example, by also reporting the validation error with a dropout rate of 0.3 in brackets.

5 RELATED WORK

While second order methods have proved to be highly effective tools for optimising deterministic functions (Wright & Nocedal, 1999; Moré, 1978) their application to stochastic optimisation, and in particular to deep neural networks remains an active area of research. Many methods have been developed to improve stochastic optimisation with curvature information to avoid slow progress in ill-conditioned regions (Dauphin et al., 2014), while avoiding the cost of storing and inverting a Hessian matrix. A popular approach is to construct updates from a buffer of parameter gradients and their first-and-second-order moments at previous iterates (*e.g.* AdaGrad (Duchi et al., 2011), AdaDelta (Zeiler, 2012), RMSProp (Tieleman & Hinton, 2012) or Adam (Kingma & Ba, 2014)). These solvers benefit from needing no additional function evaluations beyond traditional mini-batch stochastic gradient descent. Typically they set adaptive learning rates by making use of empirical estimates of the curvature with a *diagonal* approximation to the Hessian (*e.g.* Zeiler (2012)) or a rescaled diagonal Gauss-Newton approximation (*e.g.* Duchi et al. (2011)). While the diagonal structure decreases the computational cost, their overall efficiency remains limited and in many cases can be matched by a well tuned SGD solver (Wilson et al., 2017).

Second order solvers take a different approach, investing more computation per iteration in the hope of achieving higher quality updates. To achieve this higher quality, they invert the Hessian matrix H, or a tractable approximation such as the Gauss-Newton approximation (Martens, 2010; Martens & Sutskever, 2011; Botev et al., 2017) (described in section 2). Another family of approaches, which have proven effective for tasks such as classification, introduce second order information with *natural gradients* (Amari, 1998). In this context, it is common to derive a loss function from a Kullback-Leibler (KL) divergence. The natural gradient makes use of the infinitesimal distance induced by the latter to follow the curvature in the Riemannian manifold equipped with this new distance. In practice the natural gradient method amounts to replacing the Hessian H in the modified gradient formula $H^{-1}J$ with the Fisher matrix F, which facilitates traversal of the optimal path in the metric space induced by the KL-divergence. Since the seminal work of Amari (1998) several authors have studied variations of this idea. TONGA (Roux et al., 2008) relies on the empirical Fisher matrix where the previous expectation over the model predictive distribution is replaced by the sample predictive distribution. The works of Pascanu & Bengio (2013); Martens established a link between Gauss-Newton methods and the natural gradient. More recently Martens & Grosse (2015) introduced the KFAC optimiser which uses a block diagonal approximation of the Fisher matrix. This was shown to be an efficient stochastic solver in several settings, but it remains a computationally challenging approach for larger-scale deep networks problems.

Many of the methods discussed above perform an explicit system inversion that can often prove prohibitively expensive (Yang & Amari, 1998). Consequently, a number of works (Martens, 2010; Martens & Sutskever, 2011; Zhang et al., 2017) have sought to exploit the cheaper computation of Hessian-vector products via automatic differentiation (Pearlmutter, 1994; Schraudolph, 2002), to perform system inversions with conjugate gradients (Hessian-free methods). Other approaches (Carmon et al., 2018; Agarwal et al., 2017) have resorted to rank-1 approximations of the Hessian for efficiency. While these methods have had some success, they have only been demonstrated on non-deep problems of moderate scale compared to the state-of-the-art in deep learning. We speculate that the main reason they are not widely adopted is their requirement of several steps (network passes) per parameter update (Sutskever et al., 2013; Koh & Liang, 2017), which would put them at a similar disadvantage w.r.t. first-order methods as the Hessian-free method that we tested (fig. 5 in the appendix). Perhaps most closely related to our approach, Orr (1995) uses automatic differentiation to compute Hessian-vector products to construct adaptive, per-parameter learning rates.

6 CONCLUSIONS AND FUTURE WORK

In this work, we have proposed a practical second-order solver that has been specifically tailored for deep-learning-scale stochastic optimisation problems. We showed that our optimiser can be applied to a large range of datasets and reach better training error than first order method with the same number of iterations, with essentially no hyper-parameters tuning. In future work, we intend to bring more improvements to the wall-clock time of our method by engineering the FMAD operation to the same standard as back-propagation, and study optimal trust-region strategies to obtain λ in closed-form.

REFERENCES

Naman Agarwal, Brian Bullins, and Elad Hazan. Second-order stochastic optimization for machine learning in linear time. *The Journal of Machine Learning Research*, 18(1):4148–4187, 2017.

Shun-Ichi Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998.

Aleksandar Botev, Julian Hippolyt Ritter, and David Barber. Practical gauss-newton optimisation for deep learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-17)*, 2017.

Yair Carmon, John C Duchi, Oliver Hinder, and Aaron Sidford. Accelerated methods for nonconvex optimization. *SIAM Journal on Optimization*, 28(2):1751–1772, 2018.

Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Return of the devil in the details: Delving deep into convolutional nets. *arXiv preprint arXiv:1405.3531*, 2014.

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

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

Gabriel Goh. Why momentum really works. *Distill*, 2017. doi: 10.23915/distill.00006. URL `http://distill.pub/2017/momentum`.

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

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

Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *International Conference on Machine Learning*, pp. 1885–1894, 2017.

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

Nicolas Loizou and Peter Richtárik. Momentum and stochastic momentum for stochastic gradient, newton, proximal point and subspace descent methods. *arXiv preprint arXiv:1712.09677*, 2017.

James Martens. New insights and perspectives on the natural gradient method. URL `https://arxiv.org/pdf/1412.1193.pdf`.

James Martens. Deep learning via hessian-free optimization. In *ICML*, volume 27, pp. 735–742, 2010.

James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pp. 2408–2417, 2015. `https://github.com/tensorflow/kfac/`.

James Martens and Ilya Sutskever. Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 1033–1040. Citeseer, 2011.

Jorge J Moré. The levenberg-marquardt algorithm: implementation and theory. In *Numerical analysis*, pp. 105–116. Springer, 1978.

Genevieve B Orr. *Dynamics and algorithms for stochastic learning*. PhD thesis, Department of Computer Science and Engineering, Oregon Graduate Institute, Beaverton, OR, 1995.

Razvan Pascanu and Yoshua Bengio. Revisiting natural gradient for deep networks. In *International Conference on Learning Representations 2013(workshop Track)*, 2013.

Barak A Pearlmutter. Fast exact multiplication by the hessian. *Neural computation*, 6(1):147–160, 1994.

Ali Rahimi and Ben Recht. Reflections on random kitchen sinks. `http://www.argmin.net/2017/12/05/kitchen-sinks/`, 2017. Accessed: 2018-05-11.

Nicolas L. Roux, Pierre antoine Manzagol, and Yoshua Bengio. Topmoumoute online natural gradient algorithm. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis (eds.), *Advances in Neural Information Processing Systems 20*, pp. 849–856. Curran Associates, Inc., 2008.

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

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pp. 1139–1147, 2013.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. In *Advances in Neural Information Processing Systems*, pp. 4151–4161, 2017.

Stephen Wright and Jorge Nocedal. Numerical optimization. *Springer Science*, 35(67-68):7, 1999.

Howard Hua Yang and Shun-ichi Amari. Complexity issues in natural gradient descent method for training multilayer perceptrons. *Neural Computation*, 10(8):2137–2157, 1998.

Xin-She Yang and Suash Deb. Engineering optimisation by cuckoo search. *International Journal of Mathematical Modelling and Numerical Optimisation*, 1(4):330–343, 2010.

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

Huishuai Zhang, Caiming Xiong, James Bradbury, and Richard Socher. Block-diagonal hessian-free optimization for training neural networks. *arXiv preprint arXiv:1712.07296*, 2017.

A ADDITIONAL PROOFS

Theorem A.1. *Let f be a convex quadratic function, and its hyper-parameters $\beta > 0$, $\rho > 0$ satisfy*

$$\tfrac{3}{2}\beta h_{\max} - 1 < \rho < 1 + \beta h_{\min}, \tag{18}$$

where $h_{\min}$ and $h_{\max}$ are the smallest and largest eigenvalues of the Hessian H, respectively.

Then Algorithm 1 converges linearly to the minimum of f.

Corollary A.1.1. *Algorithm 1 converges for any momentum parameter $0 < \rho < 1$ with a sufficiently small learning rate $\beta > 0$, regardless of the (PSD) Hessian spectrum.*

Proof of Theorem A.1. We assume the quadratic model:

$$f(w) = \frac{1}{2} w^T H w - b^T w, \tag{19}$$

which has Hessian matrix H, and gradient $J(w) = Hw - b$.

Without loss of generality, we will consider the pure Newton method, where H is not regularized ($\lambda = 0$):[5]

$$z_{t+1} = \rho z_t - \beta(H z_t + J(w_t)) \tag{20}$$

$$w_{t+1} = w_t + z_{t+1} \tag{21}$$

eq. 20 can be rearranged to

$$z_{t+1} = (\rho I - \beta H) z_t - \beta J(w_t). \tag{22}$$

We now perform a change of variables to diagonalize the Hessian, $H = Q\mathrm{diag}(h)Q^T$, with Q orthogonal and h the vector of eigenvalues. Let $w^* = \arg\min_w f(w) = H^{-1}b$ be the optimal solution of the minimization. Then, replacing $w_t = Qx_t + w^*$ in eq. 22:

$$Qy_{t+1} = (\rho I - \beta H) Q y_t - \beta H Q x_t \tag{23}$$

with $J = H(Qx_t + w^*) - b = H(Qx_t + H^{-1}b) - b = HQx_t$.

Then, expanding H with its eigendecomposition,

$$Qy_{t+1} = \rho Q y_t - \beta Q \mathrm{diag}(h) Q^T Q y_t - \beta Q \mathrm{diag}(h) Q^T Q x_t. \tag{24}$$

Left-multiplying by Q^T,and canceling out Q due to orthogonality,

$$y_{t+1} = \rho y_t - \beta \mathrm{diag}(h) y_t - \beta \mathrm{diag}(h) x_t. \tag{25}$$

Similarly for eq. 21, replacing $z_t = Qy_t$ yields

$$x_{t+1} = x_t + y_{t+1}. \tag{26}$$

Note that each pair formed by the corresponding element of y_t and x_t is an independent system with only 2 variables, since the pairs do not interact (eq. 25 and 26 only contain element-wise operations). From now on, we will be working on the ith element of each vector.

We can thus write eq. 25 and 26 (for a single element i of each) as a vector equation:

$$\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} y_{t+1,i} \\ x_{t+1,i} \end{bmatrix} = \begin{bmatrix} \rho - \beta h_i & -\beta h_i \\ 0 & 1 \end{bmatrix} \begin{bmatrix} y_{t,i} \\ x_{t,i} \end{bmatrix}. \tag{27}$$

The matrix on the left is necessary to express the fact that the y_{t+1} factor in eq. 26 must be moved to the left-hand side, which corresponds to iteration $t+1$ ($x_{t+1} - y_{t+1} = x_t$). Left-multiplying eq. 27 by the inverse,[6]

$$\begin{bmatrix} y_{t+1,i} \\ x_{t+1,i} \end{bmatrix} = \begin{bmatrix} \rho - \beta h_i & -\beta h_i \\ \rho - \beta h_i & 1 - \beta h_i \end{bmatrix} \begin{bmatrix} y_{t,i} \\ x_{t,i} \end{bmatrix}. \tag{28}$$

[5]For the general case, the momentum parameter ρ is simply replaced by the slightly perturbed value $\rho - \beta\lambda$ (since $\rho \gg \beta\lambda$), and similar derivations follow.

[6]We have: $\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$

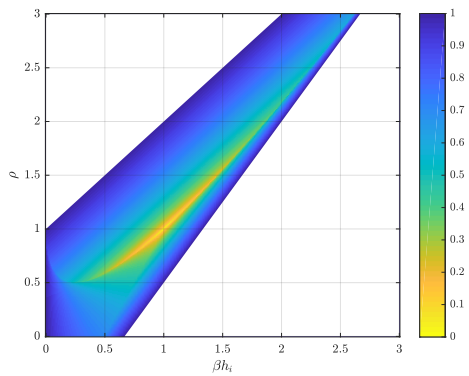

Figure 4: Convergence rate as a function of hyper-parameters ρ, β, and Hessian eigenvalue h_i. Lower values (brighter) are better. The white areas show regions of non-convergence.

This is the transition matrix R_i that characterizes the iteration, and taking its power models multiple iterations in closed-form:

$$\begin{bmatrix} y_{t,i} \\ x_{t,i} \end{bmatrix} = R_i^t \begin{bmatrix} y_{0,i} \\ x_{0,i} \end{bmatrix}. \tag{29}$$

The two eigenvalues of R_i are given in closed-form by:

$$\text{eig}\,(R_i) = \frac{1}{2}\left(\rho - 2\beta h_i + 1 \pm \sqrt{(\rho - 2\beta h_i)^2 - 2\rho + 1}\right) \tag{30}$$

The series in eq. 29 converges when $|\text{eig}\,(R_i)| < 1$ simultaneously for both eigenvalues, which is equivalent to:

$$\tfrac{3}{2}\beta h_i - 1 < \rho < 1 + \beta h_i, \tag{31}$$

with $\rho > 0$ and $\beta h_i > 0$. Note that when using the Gauss-Newton approximation of the Hessian, $h_i > 0$ and thus the last condition simplifies to $\beta > 0$.

Since eq. 31 has to be satisfied for every eigenvalue, we have

$$\tfrac{3}{2}\beta h_{\max} - 1 < \rho < 1 + \beta h_{\min}, \tag{32}$$

with $h_{\min}$ and $h_{\max}$ the smallest and largest eigenvalues of the Hessian H, respectively, proving the result.

The rate of convergence is the largest of the two values $|\text{eig}\,(R_i)|$. When the argument of the square root in eq. 30 is non-negative, it does not admit an easy interpretation; however, when it is negative, eq. 30 simplifies to:

$$|\text{eig}\,(R_i)| = \sqrt{\rho - \beta h_i}. \tag{33}$$

□

Graphical interpretation. The convergence rate for a single eigenvalue is illustrated in Figure 4. Graphically, the regions of convergence for different eigenvalues will differ only by a scale factor along the βh_i axis (horizontal stretching of Figure 4). Moreover, the largest possible range of βh_i values is obtained when $\rho = 1$, and that range is $0 < \beta h_i < \frac{4}{3}$. We can infer that the intersection of the regions of convergence for several eigenvalues will be maximized with $\rho = 1$, for any fixed β.

Theorem A.2. *Let the Hessian $\hat{H}_{t+1}$ be positive definite (which holds when the objective is convex or when Gauss-Newton approximation and trust region are used). Then the update z_{t+1} in Algorithm 1 is a descent direction when β and ρ are chosen according to eq. 16, and $z_{t+1} \neq 0$.*

Proof. To show that the update represents a descent direction, it suffices to show that $J^T z_{t+1} < 0$ (where we have written $J = J(w_t)$ to simplify notation). Since the surrogate Hessian $\hat{H}_{t+1}$ is positive definite (PD) by construction, the update $z_{t+1} = \rho z_t - \beta \Delta_{z_{t+1}}$ satisfies $z_{t+1}^T \hat{H}_{t+1} z_{t+1} > 0$. It is therefore sufficient to prove that $J^T z_{t+1} + z_{t+1}^T \hat{H}_{t+1} z_{t+1} \leq 0$.

It follows from their definition in eq. (16) that ρ and β minimise the RHS of

$$J^T z_{t+1} + \frac{1}{2} z_{t+1}^T \hat{H}_{t+1} z_{t+1} =$$
$$\begin{bmatrix} J^T \Delta_{z_{t+1}} \\ J^T z_t \end{bmatrix}^T \begin{bmatrix} -\beta \\ \rho \end{bmatrix} + \frac{1}{2} \begin{bmatrix} -\beta \\ \rho \end{bmatrix}^T \begin{bmatrix} \Delta_{z_{t+1}}^T \hat{H}_{t+1} \Delta_{z_{t+1}} & z_t^T \hat{H}_{t+1} \Delta_{z_{t+1}} \\ z_t^T \hat{H}_{t+1} \Delta_{z_{t+1}} & z_t^T \hat{H}_{t+1} z_t \end{bmatrix} \begin{bmatrix} -\beta \\ \rho \end{bmatrix} \tag{34}$$

In particular, they minimise a quadratic form in $(-\beta, \rho)$ with the following symmetric hessian

$$K = \begin{bmatrix} \Delta_{z_{t+1}}^T \hat{H}_{t+1} \Delta_{z_{t+1}} & z_t^T \hat{H}_{t+1} \Delta_{z_{t+1}} \\ z_t^T \hat{H} \Delta_{z_{t+1}} & z_t^T \hat{H}_{t+1} z_t \end{bmatrix}. \tag{35}$$

Moreover, for any $x = (x_1, x_2) \in \mathbb{R}^2$,

$$\begin{aligned} x^T K x &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} \Delta_{z_{t+1}}^T \hat{H}_{t+1} \Delta_{z_{t+1}} & z_t^T \hat{H}_{t+1} \Delta_{z_{t+1}} \\ z_t^T \hat{H} \Delta_{z_{t+1}} & z_t^T \hat{H}_{t+1} z_t \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= (x_1 \Delta_{z_{t+1}} + x_2 z_t)^T \hat{H} (x_1 \Delta_{z_{t+1}} + x_2 z_t). \end{aligned} \tag{36}$$

Consequently, K is guaranteed to be Positive Semidefinite (PSD) and the form is convex with zero gradient at the minimum. Since $z_{t+1} \neq 0$, it follows that at least one of the following holds: (1) K is invertible and hence PD (rather than simply PSD); (2) one of factors $z_t = 0$ or $\Delta_{z_{t+1}} = 0$ is zero; (3) the factors $z_t = 0$ and $\Delta_{z_{t+1}} = 0$ are colinear. In the first case we have,

$$\begin{aligned} J^T z_{t+1} + \frac{1}{2} z_{t+1}^T \hat{H}_{t+1} z_{t+1} &= \\ -\frac{1}{2} \begin{bmatrix} J^T \Delta_{z_{t+1}} \\ J^T z_t \end{bmatrix}^T & \begin{bmatrix} \Delta_{z_{t+1}}^T \hat{H}_{t+1} \Delta_{z_{t+1}} & z_t^T \hat{H}_{t+1} \Delta_{z_{t+1}} \\ z_t^T \hat{H}_{t+1} \Delta_{z_{t+1}} & z_t^T \hat{H}_{t+1} z_t \end{bmatrix}^{-1} \begin{bmatrix} J^T \Delta_{z_{t+1}} \\ J^T z_t \end{bmatrix} \end{aligned} \tag{37}$$

Since the inverse of a PD matrix is PD, the RHS of eq. (34) is negative. Further, as $\hat{H}_{t+1}$ is PD, it follows that final term in eq. (34) is positive, thus K is PD, showing that $J^T z_{t+1} < 0$.

For the second case in which $z_t = 0$ or $\Delta_{z_{t+1}} = 0$, the system reduces to a trivial convex second order equation in ρ or β with a negative solution.

Finally, consider the case when z_t and $\Delta_{z_{t+1}}$ are colinear but both non-negative. Writing $\Delta_{z_{t+1}} = \alpha z_t$ for $\alpha \in \mathbb{R}$, we note that at the minimum we have

$$J^T z_{t+1} + \frac{1}{2} z_{t+1}^T \hat{H}_{t+1} z_{t+1} = -\frac{1}{2} \begin{bmatrix} J^T \Delta_{z_{t+1}} \\ J^T z_t \end{bmatrix}^T \begin{bmatrix} -\beta \\ \rho \end{bmatrix} = -\frac{1}{2} (\rho - \alpha\beta) J^T z_t. \tag{38}$$

Thus at the minimum (38) is negative, closing the proof. □

Remark. *It follows from the definition of ρ and β that if $J(w_t) = 0$, then $z_{t+1} = 0$.*

Remark. *If $z_{t+1} = 0$, then $z_{t+2} = -\beta J(w_{t+1})$, i.e. we reset the momentum variable z. This guarantees that the algorithm takes a strictly descending direction at least every two steps.*

B ADDITIONAL RESULTS AND IMPLEMENTATION DETAILS

B.1 CONFIGURATIONS FOR SMALL-SCALE DATASET EXPERIMENTS

Here we provide additional details of the small-scale datasets described in sec. 4. As noted in the main paper, to give every method the best chance of working effectively we first perform a grid-search over its hyperparameters. This search is performed for each of the small-scale dataset experiments. For each first order solver listed in Table. 1, we select the configuration which achieves the lowest average error across the final ten iterations of a trajectory. The values included in the search were:

- SGD with momentum: learning rates: Γ, momentum values: $0.9, 0.95, 0.99$
- Adam: learning rates Γ, $\beta_1 : 0.9, 0.99$, $\beta_2 : 0.99, 0.999$

where $\Gamma = 0.1, 0.05, 0.01, 0.05, 0.001, 0.005, 0.0001, 0.0005$.

B.2 RANDOM ARCHITECTURE EXPERIMENT SETUP

Each optimiser is tested on 50 random networks, that are held fixed across all methods. The number of convolutional layers is uniformly sampled between 3 and 10, and the number of channels in each layer is drawn uniformly, in powers of two, between 32 and 256. The kernel size is 3×3. Following each convolution (except the last one) there is a ReLU activation and batch-normalisation, and 3×3 max-pooling (stride 2) is placed with 50% chance. Training and evaluation is performed on CIFAR10, with a batch size of 256.

B.3 WALL-CLOCK TIME RESULTS WITH CONJUGATE GRADIENT

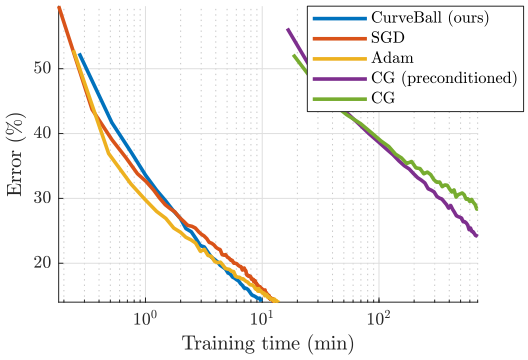

Figure 5: Training error vs. wall clock time (basic CIFAR-10 model). The time axis is logarithmic to show a comparison with conjugate-gradient-based Hessian-free optimisation. Due to the CG iterations, it takes an order of magnitude more time to converge than first-order solvers and our proposed second-order solver, despite the efficient GPU implementation.