Small Steps and Giant Leaps: Minimal Newton Solvers for Deep Learning

João F. Henriques Sebastien Ehrhardt Samuel Albanie Andrea Vedaldi
Visual Geometry Group, University of Oxford
{joao,hyenal,albanie,vedaldi}@robots.ox.ac.uk

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 (**?**) 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 (**?**).

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, it soon 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 (**?**, p. 27), and make provably better progress in the regions where gradient descent stalls. 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 find an optimum (or at least a stationary point) of 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]

$$z \leftarrow \rho z - \beta J(w) \tag{2}$$
$$w \leftarrow w + z \tag{3}$$

with a momentum parameter ρ. Momentum GD, as given by eq. 2-3, can be shown to have faster convergence than GD for convex functions, remaining stable under higher learning rates, and exhibits somewhat better resistance to poor scaling of the objective function (**??**). 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 with a parameter $\lambda > 0$, obtaining what is known as the *Levenberg* (**?**) 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).

3. Method

3.1. 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*. Note that, because the loss is a scalar function, the starting projection vector v typically used in gradient descent is a scalar and we set $v = 1$. For intermediate computations, however, it is generally a (vectorized) tensor of gradients (see eq. 9).

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

[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),

Algorithm 1. CURVEBALL (proposed).

$z_0 = \mathbf{0}$
for $t = 0, ..., T-1$ **do**
  $\Delta_z = \hat{H}(w_t) z_t + J(w_t)$
  $z_{t+1} = \rho z_t - \beta \Delta_z$
  $w_{t+1} = w_t + z_{t+1}$
end for

Algorithm 2. Simplified Hessian-free method.

for $t = 0, ..., T-1$ **do**
  $z_0 = -J(w_t)$
  for $r = 0, ..., R-1$ (or convergence) **do**
    $z_{r+1} = \mathrm{CG}(z_r, \hat{H}(w_t) z_r, J(w_t))$
  end for
  $w_{t+1} = w_t + z_R$
end for

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 $a \circ b \circ c$,

$$\overrightarrow{\mathrm{AD}}_{a \circ b \circ c}(v) = ((v J_a) J_b) J_c \quad (8)$$

$$\overleftarrow{\mathrm{AD}}_{a \circ b \circ c}(v') = J_a (J_b (J_c v')) \quad (9)$$

Because of this, both operations have similar computational overhead, and can be implemented similarly.

3.2. 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 a surrogate matrix with this property, which prevents second-order methods from being attracted to saddle-points (**?** discusses this problem). One of the most widely used is the Gauss-Newton matrix (**??**, p. 254):

$$\hat{H} = J_\phi H_L J_\phi^T, \quad (10)$$

where H_L is the Hessian of the loss function. 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. For the method that we propose, and indeed for any method that implicitly inverts the Hessian (or its approximation), only computing Hessian-vector products $\hat{H}v$ is required. As such, eq. 10 takes a very convenient form:

$$\hat{H}v = J_\phi \left(H_L \left(J_\phi^T v \right) \right) \quad (11)$$

$$= \overleftarrow{\mathrm{AD}}_\phi \left(H_L \left(\overrightarrow{\mathrm{AD}}_\phi(v) \right) \right). \quad (12)$$

The cost of eq. 12 is thus equivalent to that of two back-propagation 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.

We remark that this builds on a classic result (**?**), however with two major differences. First, their method computes products with the exact Hessian H, while we use the Gauss-Newton matrix $\hat{H}$, to be robust to saddle-points. Second, we leverage modern automatic differentiation tools, while they rely on a symbolic manipulation technique which requires new equations to be derived for each model change.

3.3. Fast second-order optimisation

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:

$$f(w+z) \simeq \hat{f}(w, z) = f(w) + z^T J(w) + \tfrac{1}{2} z^T H(w) z \quad (13)$$

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

$$z = \arg\min_{z'} \hat{f}(z') = \arg\min_{z'} \tfrac{1}{2} z'^T \hat{H} z' + z'^T J \quad (14)$$

In general, a step z is found by minimizing eq. 14, either via explicit inversion $\hat{H}^{-1} J$ (**??**) or the conjugate-gradient (CG) method (**?**). The later approach, called the Hessian-free method (also Truncated Newton or Newton-CG (**?**, p. 168)) is the most economical in terms of memory, since it only needs access to Hessian-vector products (section 3.2). 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). Note 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.

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 all theoretical guarantees, which were proved assuming that the starting point of the CG iterations is always $z_0 = -J(w_t)$ (**?**, p. 124). 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. 14 w.r.t. z yields:

$$\triangle_z = J_{\hat{f}(z)} = \hat{H}z + J \tag{15}$$

Applying these changes to the Hessian-free method (Algorithm 2) results in Algorithm 1. We also introduced an optional factor ρ that decays z each step (Algorithm 1, line 4). The practical reason for its inclusion is to gradually forget stale updates, made necessary by the non-linearity of f. More formally, this change is exactly equivalent to adding a regularization term $(1-\rho)\left\|z\right\|^2$, which vanishes when $\rho \simeq 1$ (as usually recommended for momentum parameters (**?**)).

Surprisingly, despite being derived from a second-order method, we can contrast Algorithm 1 to momentum GD (eq. 2-3) and see that they are almost equivalent, except for an extra curvature term $\hat{H}(w)z$. 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 3.2, it is easy to implement eq. 15, including a regularization term λI (section 2.1). We can further improve eq. 15 by grouping the operations to minimize the number of automatic differentiation (back-propagation) steps:

$$\begin{aligned} \triangle_z &= \left(J_\phi H_L J_\phi^T + \lambda I\right) z + J_\phi J_L & (16) \\ &= J_\phi \left(H_L J_\phi^T z + J_L\right) + \lambda z & (17) \end{aligned}$$

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 3.1).

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. 14 allows us to draw on standard results in optimisation. At any given step, the optimal ρ and β can be obtained by solving a 2×2 linear system (**?**, sec. 7):

$$\begin{bmatrix} \beta \\ -\rho \end{bmatrix} = \begin{bmatrix} \Delta_z^T \hat{H} \Delta_z & z^T \hat{H} \Delta_z \\ z^T \hat{H} \Delta_z & z^T \hat{H} z \end{bmatrix}^{-1} \begin{bmatrix} J^T \Delta_z \\ J^T z \end{bmatrix} \tag{18}$$

Note that, in calculating the proposed update (eq. 16), 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. A detailed proof can be found in Appendix A.1.

Automatic λ hyper-parameter rescaling. The regularization term λI (eq. 4) can be interpreted as a trust-region (**?**, p. 68). 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 (**?**, p. 69), 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. We show the evolution of automatically tuned hyper-parameters in Appendix B.2.

Convergence proofs. 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 (**?**, 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 non-quadratic functions, CURVEBALL's steps are always in a descent direction, when using the automatic hyper-parameter tuning of eq. 18 (**Theorem A.2**). We note

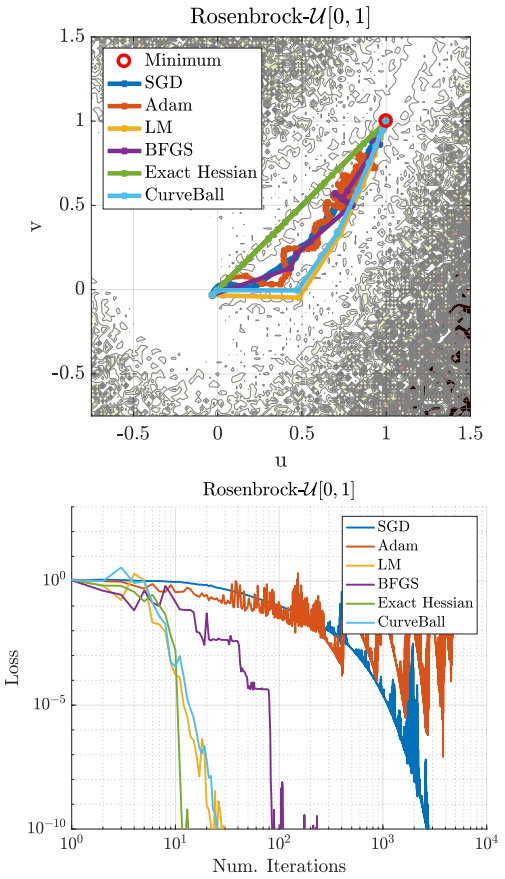

Figure 1. **Degenerate optimisation problems with known solutions**. Top: Trajectories on the Stochastic Rosenbrock function for different solvers (higher function values have darker shades). Bottom: Evolution of the loss per iterations for the same trajectories.

Table 1. **Optimiser comparison on small degenerate datasets.** For each optimiser, we report the mean±std of the number of to iterates reach the solution. The limits of the uniform distribution used in the stochastic Rosenbrock function (eq. 19) are $\mathcal{U}[\lambda_1, \lambda_2]$.

	Deterministic	Rosenbrock $\mathcal{U}[0,1]$	$\mathcal{U}[0,3]$	Raihimi & Recht
SGD + momentum	370 ± 40	846 ± 217	4069 ± 565	95 ± 2
Adam (**?**)	799 ± 160.5	1290 ± 476	2750 ± 257	95 ± 5
Lev.-Marquardt (**?**)	16 ± 4	14 ± 3	17 ± 4	9 ± 4
BFGS (**?**, p. 136)	19 ± 4	44 ± 21	63 ± 29	43 ± 21
Exact Hessian	14 ± 1	$\mathbf{10 \pm 3}$	17 ± 4	$\mathbf{9 \pm 0.5}$
CURVEBALL (ours)	$\mathbf{13 \pm 0.5}$	12 ± 1	$\mathbf{13 \pm 1}$	35 ± 11

that due to the Gauss-Newton approximation and the trust region (eq. 4) this is always verified in practice.

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 architectures, we begin by applying it 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 (**?**)[3], as well as with more traditional methods such as Levenberg-Marquardt, BFGS (**?**, p. 136) (with cubic line-search) and Newton's method with the exact Hessian. 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 (**?**), $\mathcal{R} : \mathbb{R}^2 \rightarrow \mathbb{R}$:

$$\mathcal{R}(u, v) = (1 - u)^2 + 100\epsilon_i(v - u^2)^2, \qquad (19)$$

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 **?**. 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 suppl.). 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 Newton method with an exact Hessian[4] generally performs best, followed closely by Levenberg-Marquardt (LM), however they are impractical for larger-scale problems. Our method delivers comparable (and sometimes better) 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, CURVEBALL, Newton and LM converge in few iterations.

[3] We also experimented with RMSProp (**?**), AdaGrad (**?**) and AdaDelta (**?**), but found that these methods consistently underperformed Adam and SGD on these "toy" problems.

[4] When the Hessian contains negative eigenvalues, their absolute values are used (**?**).

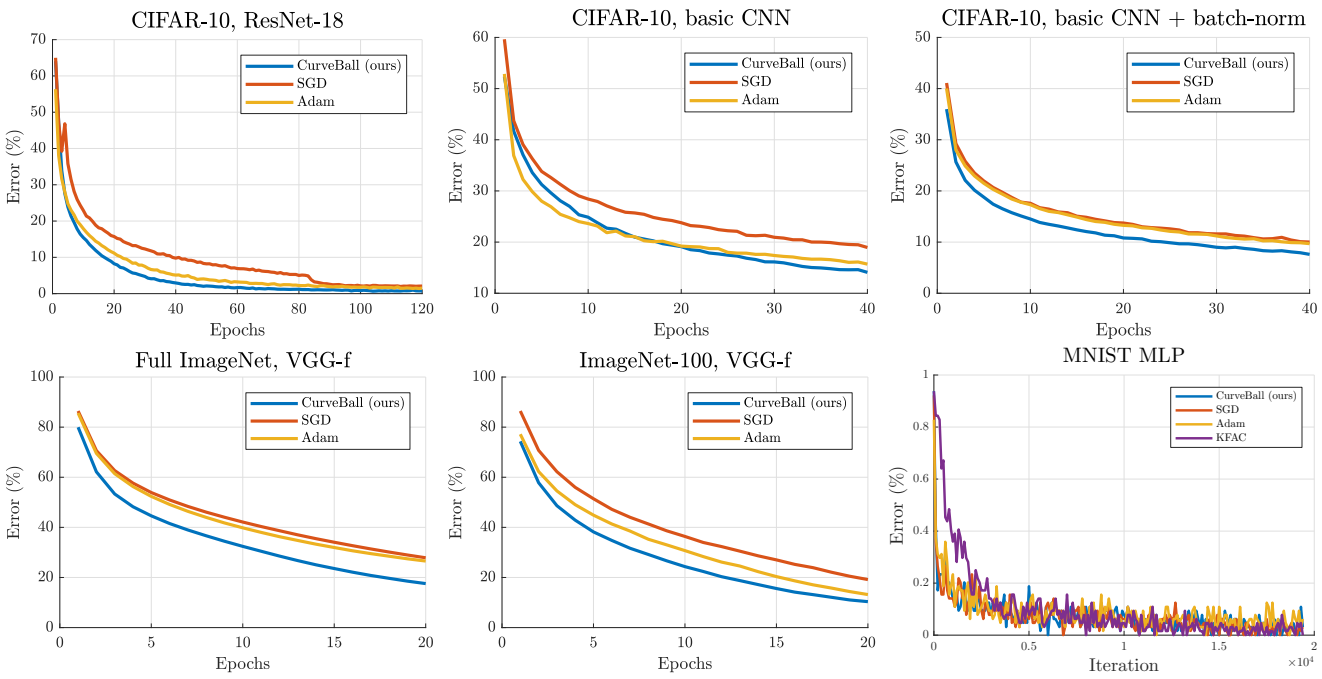

Figure 2. **Comparison with the different optimisers for various datasets and networks**. Training error is shown, as it is the quantity being optimised. CURVEBALL performs well under a variety of realistic settings, including large-scale datasets (ImageNet), batch-normalisation, and severely over-parameterised models (ResNet). **Its hyper-parameters are the same** except for the top-center panel (no batch-norm).

CIFAR. We now turn to the task of training deep networks on more realistic datasets. Classic second-order methods are 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).[5] We train this network for 20 epochs on CIFAR-10, with and without batch-normalisation (which is known to improve conditioning (**?**)) 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 (**?**). 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 (**?**). 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 256 and follow the settings described by **?**. 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.

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 (**?**). 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.3 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 25$^{\text{th}}$-75$^{\text{th}}$ percentiles (shaded regions). CURVEBALL consistently outperforms first-order methods, with the bulk of the achieved errors below those of SGD and Adam.

[5]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 (train./val.) for different models and optimisation methods. CURVEBALL λ uses automatic λ rescaling (sec. 3.3). Numbers in bracket show validation error with 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**	19.2 / 39.8
Adam	15.7 / 19.7	9.6 / 16.1	1.4 / 14.0	13.2 / 35.9

Comparison to other second-order methods. We used the public implementation of the second-order KFAC (**?**) method and tested their proposed scenario of a 4-layer MLP (with output sizes 128-64-32-10) with hyperbolic tangent activations for MNIST classification. 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.

A major advantage of our method is its minimal implementation based on standard deep learning tools, allowing us to easily use improvements such as batch-normalisation. To highlight this fact, in Fig. 3 (right) we show again the basic CIFAR setting, this time comparing CURVEBALL with and without batch-norm with KFAC, for which such an addition would be relatively non-trivial to implement. This shows that the combination of our method and batch-norm is more powerful than each in isolation, similarly to what happens with first-order methods (Fig. 2).

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, we observe that our method is competitive with first-order solvers, while not requiring *any* tuning. Moreover, our prototype implementation includes 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) (**?**). We show a comparison in logarithmic time in Appendix B.4. 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 3.3).

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 in 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 (**??**, p. 164) 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 (**?**), 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 (**?**), AdaDelta (**?**), RMSProp (**?**) or Adam (**?**)). 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.* **?**) or a rescaled diagonal Gauss-Newton approximation (*e.g.* **?**). While the diagonal structure reduces computation, their overall efficiency remains limited and in many cases can be matched by a well tuned SGD solver (**?**).

Second-order solvers take a different approach, investing more computation per iteration in the hope of achieving higher quality updates. Trust-region methods (**?**) and cubic regularization (**??**) are canonical examples. To achieve this higher quality, they invert the Hessian matrix H, or a tractable approximation such as the Gauss-Newton approximation (**???**) (described in section 2), or other regularized (**?**) or subsampled versions of the Hessian (**??**). Another line of work belonging to the trust-region family (**?**), which has proven effective for tasks such as classification, introduces second order information with *natural gradients* (**?**). While it implements a trajectory on a Riemannian manifold derived from a Kullback-Leibler (KL) divergence, in practice it amounts to replacing the Hessian H in the modified gradient formula $H^{-1}J$ with the Fisher matrix F. Since the seminal work of **?** several authors have studied variations of this idea. TONGA (**?**) 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 **?** and **?** established a link between Gauss-Newton methods and the natural gradient. More recently **?** 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 remains

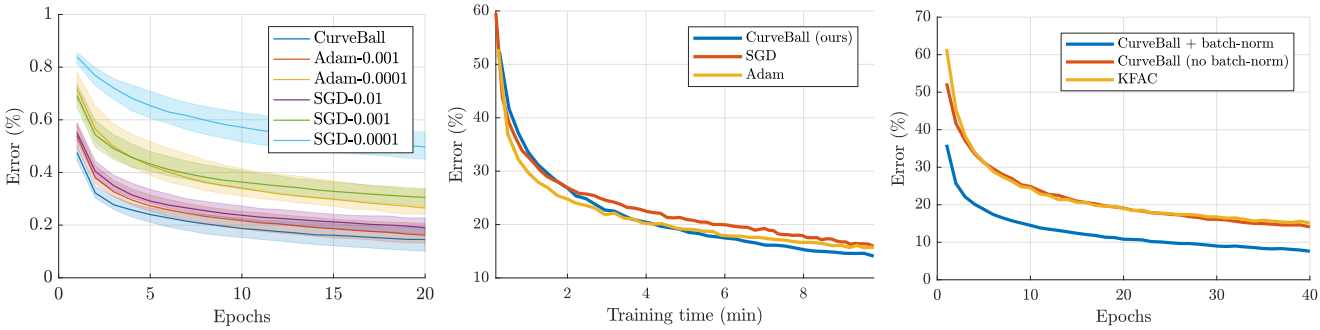

Figure 3. (Left) **Results of 50 randomly-generated architectures on CIFAR10.** The median (thick lines) and 25$^{\text{th}}$-75$^{\text{th}}$ percentiles (shaded region) are shown. Numbers in the legend represent learning rates (fixed for CURVEBALL). (Center) **Training error vs. wall clock time** (basic CIFAR-10 model). (Right) **Batch-normalisation** is easily combined with our method, which is not the case with KFAC and others.

computationally challenging. In particular, it requires an order of magnitude more memory than the parameters themselves, which may be infeasible in some cases. In contrast, ours has the same requirements as momentum-SGD.

Many of the methods discussed above perform an explicit system inversion that can often prove prohibitively expensive (**?**). Consequently, a number of works (**???**) have sought to exploit the cheaper computation of Hessian-vector products via automatic differentiation (**??**), to perform system inversions with conjugate gradients (Hessian-free methods). Other approaches (**??**) have resorted to rank-1 approximations of the Hessian for efficiency. While these methods have had some success, they have only been demonstrated on single-layer models 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 (**??**), which would put them at a similar disadvantage w.r.t. first-order methods as the Hessian-free method that we tested (Appendix B.4). Perhaps more closely related to our approach, **?** uses automatic differentiation to compute Hessian-vector products to construct adaptive, per-parameter learning rates.

The closest method is LiSSA (**?**), built around the idea of approximating the Hessian inverse with a Taylor series expansion. This series can be implemented as the recursion $H_{(r)}^{-1} = I + (I - H)H_{(r-1)}^{-1}$, starting with $H_{(0)}^{-1} = I$. Since LiSSA is a type of Hessian-free method, the core of the algorithm is similar to Algorithm 2: it also refines an estimate of the Newton step iteratively, but with a different update rule in line 4. With some simple algebraic manipulations, we can use the Taylor recursion to write this update in a form that is similar to ours: $z_{r+1} = z_r - \hat{H}z_r - J$. This looks similar to our gradient-descent-based update with a learning rate of $\beta = 1$ (Alg. 1, lines 3-4), with some key differences. First, they reset the state of the step estimate for every mini-batch (Alg. 2, line 2). Reusing past solutions, like momentum-SGD, is an important factor in the performance of our algorithm, since we only have to perform one update per mini-batch. In contrast, **?** report a typical number of inner-loop updates (R in Alg. 2) equal to the number of samples (e.g. $R = 10,000$ for a tested subset of MNIST). While this is not a problem for their tested case of linear Support Vector Machines, since each update only requires one inner-product, the same does not apply to deep neural networks. Second, they invert the Hessian independently for each mini-batch, while our method aggregates the implicit Hessian across all past mini-batches (with a forgetting factor of ρ). Since batch sizes are orders of magnitude smaller than the number of parameters (e.g. 256 samples vs. 60 million parameters for the VGG-f), the Hessian matrix for a mini-batch is a poor substitute for the Hessian of the full dataset in these problems, and severely ill-conditioned. Finally, while they demonstrate improved performance on convex problems with linear models, we focus on the needs of training deep networks on large datasets (millions of samples and parameters), on which no previous Newton method has been able to surpass the first-order methods that are commonly used by the deep learning community.

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 range of datasets and reach better training error than first order methods with the same number of iterations, with essentially *no hyper-parameter 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.

Acknowledgments. We are grateful to ERC StG IDIU-638009, IDIU-677195, EP/R03298X/1, DFR05420 and EPSRC AIMS CDT for support.