Practical Gauss-Newton Optimisation for Deep Learning

Aleksandar Botev [1] **Hippolyt Ritter** [1] **David Barber** [1 2]

Abstract

We present an efficient block-diagonal approximation to the Gauss-Newton matrix for feedforward neural networks. Our resulting algorithm is competitive against state-of-the-art first-order optimisation methods, with sometimes significant improvement in optimisation performance. Unlike first-order methods, for which hyperparameter tuning of the optimisation parameters is often a laborious process, our approach can provide good performance even when used with default settings. A side result of our work is that for piecewise linear transfer functions, the network objective function can have no differentiable local maxima, which may partially explain why such transfer functions facilitate effective optimisation.

1. Introduction

First-order optimisation methods are the current workhorse for training neural networks. They are easy to implement with modern automatic differentiation frameworks, scale to large models and datasets and can handle noisy gradients such as encountered in the typical mini-batch setting (Polyak, 1964; Nesterov, 1983; Kingma & Ba, 2014; Duchi et al., 2011; Zeiler, 2012). However, a suitable initial learning rate and decay schedule need to be selected in order for them to converge both rapidly and towards a good local minimum. In practice, this usually means many separate runs of training with different settings of those hyperparameters, requiring access to either ample compute resources or plenty of time. Furthermore, pure stochastic gradient descent often struggles to escape 'valleys' in the error surface with largely varying magnitudes of curvature, as the first derivative does not capture this information (Dauphin et al., 2014; Martens & Sutskever, 2011). Modern alternatives, such as ADAM (Kingma & Ba, 2014), combine the gradients at the current setting of the parameters with various heuristic estimates of the curvature from previous gradients.

Second-order methods, on the other hand, perform updates of the form $\delta = H^{-1}g$, where H is the Hessian or some approximation thereof and g is the gradient of the error function. Using curvature information enables such methods to make more progress per step than techniques relying solely on the gradient. Unfortunately, for modern neural networks, explicit calculation and storage of the Hessian matrix is infeasible. Nevertheless, it is possible to efficiently calculate Hessian-vector products Hg by use of extended Automatic Differentiation (Schraudolph, 2002; Pearlmutter, 1994); the linear system $g = Hv$ can then be solved for v, *e.g.* by using conjugate gradients (Martens, 2010; Martens & Sutskever, 2011). Whilst this can be effective, the number of iterations required makes this process uncompetitive against simpler first-order methods (Sutskever et al., 2013).

In this work, we make the following contributions:

– We develop a recursive block-diagonal approximation of the Hessian, where each block corresponds to the weights in a single feedforward layer. These blocks are Kronecker factored and can be efficiently computed and inverted in a single backward pass.

– As a corollary of our recursive calculation of the Hessian, we note that for networks with piecewise linear transfer functions the error surface has no differentiable strict local maxima.

– We discuss the relation of our method to KFAC (Martens & Grosse, 2015), a block-diagonal approximation to the Fisher matrix. KFAC is less generally applicable since it requires the network to define a probabilistic model on its output. Furthermore, for non-exponential family models, the Gauss-Newton and Fisher approaches are in general different.

– On three standard benchmarks we demonstrate that (without tuning) second-order methods perform competitively, even against well-tuned state-of-the-art first-order methods.

[1]University College London, London, United Kingdom [2]Alan Turing Institute, London, United Kingdom. Correspondence to: Aleksandar Botev <a.botev@cs.ucl.ac.uk>.

Proceedings of the 34^{th} International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017.

2. Properties of the Hessian

As a basis for our approximations to the Gauss-Newton matrix, we first describe how the diagonal Hessian blocks of feedforward networks can be recursively calculated. Full derivations are given in the supplementary material.

2.1. Feedforward Neural Networks

A feedforward neural network takes an input vector $a_0 = x$ and produces an output vector h_L on the final (L^{th}) layer of the network:

$$h_\lambda = W_\lambda a_{\lambda-1}; \quad a_\lambda = f_\lambda(h_\lambda) \qquad 1 \leq \lambda < L \quad (1)$$

where h_λ is the pre-activation in layer λ and a_λ are the activation values; W_λ is the matrix of weights and f_λ the elementwise transfer function[1]. We define a loss $E(h_L, y)$ between the output h_L and a desired training output y (for example squared loss $(h_L - y)^2$) which is a function of all parameters of the network $\theta = \left[\text{vec}\,(W_1)^\intercal, \text{vec}\,(W_2)^\intercal, \ldots, \text{vec}\,(W_L)^\intercal\right]^\intercal$. For a training dataset with empirical distribution $p(x, y)$, the total error function is then defined as the expected loss $\bar{E}(\theta) = \mathbb{E}\left[E\right]_{p(x,y)}$. For simplicity we denote by $E(\theta)$ the loss for a generic single datapoint (x, y).

2.2. The Hessian

A central quantity of interest in this work is the parameter Hessian, H, which has elements:

$$[H]_{ij} = \frac{\partial^2}{\partial\theta_i \partial\theta_j} E(\theta) \quad (2)$$

The expected parameter Hessian is similarly given by the expectation of this equation. To emphasise the distinction between the expected Hessian and the Hessian for a single datapoint (x, y), we also refer to the single datapoint Hessian as the *sample* Hessian.

2.2.1. Block Diagonal Hessian

The full Hessian, even of a moderately sized neural network, is computationally intractable due to the large number of parameters. Nevertheless, as we will show, blocks of the sample Hessian can be computed efficiently. Each block corresponds to the second derivative with respect to the parameters W_λ of a single layer λ. We focus on these blocks since the Hessian is in practice typically block-diagonal dominant (Martens & Grosse, 2015).

The gradient of the error function with respect to the weights of layer λ can be computed by recursively applying the chain rule:[2]

$$\frac{\partial E}{\partial W^\lambda_{a,b}} = \sum_i \frac{\partial h^\lambda_i}{\partial W^\lambda_{a,b}} \frac{\partial E}{\partial h^\lambda_i} = a^{\lambda-1}_b \frac{\partial E}{\partial h^\lambda_a} \quad (3)$$

Differentiating again we find that the sample Hessian for layer λ is:

$$[H_\lambda]_{(a,b),(c,d)} \equiv \frac{\partial^2 E}{\partial W^\lambda_{a,b} \partial W^\lambda_{c,d}} \quad (4)$$

$$= a^{\lambda-1}_b a^{\lambda-1}_d \left[\mathcal{H}_\lambda\right]_{a,c} \quad (5)$$

where we define the *pre-activation* Hessian for layer λ as:

$$\left[\mathcal{H}_\lambda\right]_{a,b} = \frac{\partial^2 E}{\partial h^\lambda_a \partial h^\lambda_b} \quad (6)$$

We can re-express (5) in matrix form for the sample Hessian of W_λ:

$$H_\lambda = \frac{\partial^2 E}{\partial \text{vec}\,(W_\lambda) \partial \text{vec}\,(W_\lambda)} = \left(a_{\lambda-1} a^\intercal_{\lambda-1}\right) \otimes \mathcal{H}_\lambda \quad (7)$$

where $\otimes$ denotes the Kronecker product[3].

2.2.2. Block Hessian Recursion

In order to calculate the sample Hessian, we need to evaluate the pre-activation Hessian first. This can be computed recursively as (see Appendix A):

$$\mathcal{H}_\lambda = B_\lambda W^\intercal_{\lambda+1} \mathcal{H}_{\lambda+1} W_{\lambda+1} B_\lambda + D_\lambda \quad (8)$$

where we define the diagonal matrices:

$$B_\lambda = \text{diag}\left(f'_\lambda(h_\lambda)\right) \quad (9)$$

$$D_\lambda = \text{diag}\left(f''_\lambda(h_\lambda) \frac{\partial E}{\partial a_\lambda}\right) \quad (10)$$

and f'_λ and f''_λ are the first and second derivatives of f_λ respectively.

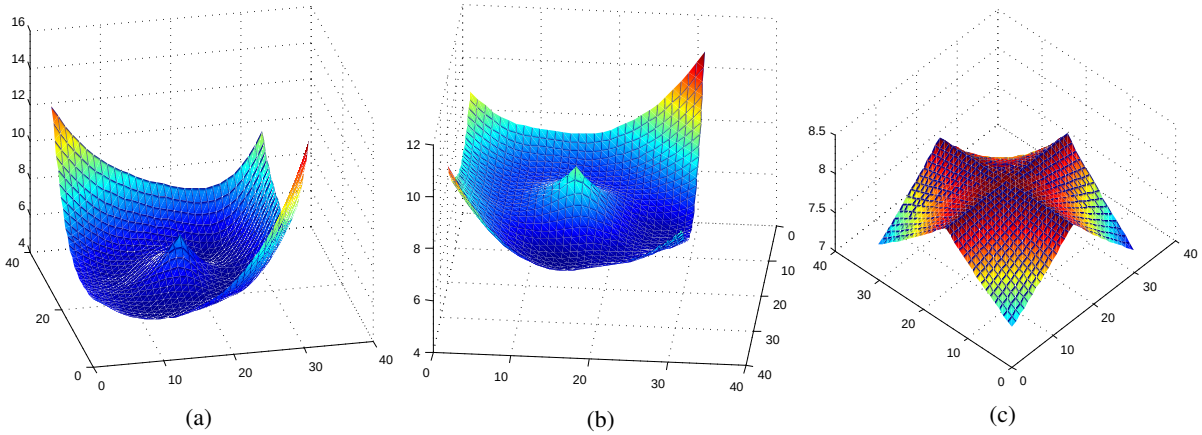
The recursion is initialised with $\mathcal{H}_L$, which depends on the objective function $E(\theta)$ and is easily calculated analytically for the usual objectives[4]. Then we can simply apply the recursion (8) and compute the pre-activation Hessian for each layer using a single backward pass through the network. A similar observation is given in (Schaul et al., 2013), but restricted to the diagonal entries of the Hessian

[1] The usual bias b_λ in the equation for h_λ is absorbed into W_λ by appending a unit term to every $a_{\lambda-1}$.

[2] Generally we use a Greek letter to indicate a layer and a Roman letter to denote an element within a layer. We use either sub- or super-scripts wherever most notationally convenient and compact.

[3] Using the notation $\{\cdot\}_{i,j}$ as the i, j matrix block, the Kronecker Product is defined as $\{A \otimes B\}_{i,j} = a_{ij} B$.

[4] For example for squared loss $(y - h_L)^2/2$, the pre-activation Hessian is simply the identity matrix $\mathcal{H}_L = I$.



(a) (b) (c)

Figure 1. Two layer network with ReLU and square loss. (a) The objective function E as we vary $W_1(x, y)$ along two randomly chosen direction matrices U and V, giving $W_1(x, y) = xU + yV$, $(x, y) \in \mathbb{R}^2$. (b) E as a function of two randomly chosen directions within W_2. (c) E for varying jointly $W_1 = xU$, $W_2 = yV$. The surfaces contain no smooth local maxima.

rather than the more general block-diagonal case. Given the pre-activation Hessian, the Hessian of the parameters for a layer is given by (7). For more than a single datapoint, the recursion is applied per datapoint and the parameter Hessian is given by the average of the individual sample Hessians.

2.2.3. No Differentiable Local Maxima

In recent years piecewise linear transfer functions, such as the ReLU function$f(x) = \max(x, 0)$, have become popular[5]. Since the second derivative f'' of a piecewise linear function is zero everywhere, the matrices D_λ in (8) will be zero (away from non-differentiable points).

It follows that if $\mathcal{H}_L$ is Positive Semi-Definite (PSD), which is the case for the most commonly used loss functions, the pre-activation matrices are PSD for every layer. A corollary is that if we fix all of the parameters of the network except for W_λ the objective function is locally convex with respect to W_λ wherever it is twice differentiable. Hence, there can be no local maxima or saddle points of the objective with respect to the parameters within a layer[6]. Note that this does not imply that the objective is convex everywhere with respect to W_λ as the surface will contain ridges along which it is not differentiable, corresponding to boundary points where the transfer function changes regimes, see Figure 1(c).

As the trace of the full Hessian H is the sum of the traces of the diagonal blocks, it must be non-negative and thus it is not possible for all eigenvalues to be simultaneously negative. This implies that for feedforward neural networks with piecewise linear transfer functions there can be no differentiable local maxima - that is, outside of pathological constant regions, all maxima (with respect to the full parameter set θ) must lie at the boundary points of the non-linear activations and be 'sharp', see Figure 1. Additionally, for transfer functions with zero gradient $f' = 0$, $\mathcal{H}_\lambda$ will have lower rank than $\mathcal{H}_{\lambda+1}$, reducing the curvature information propagating from the output layer back up the network. This suggests that it is advantageous to use piecewise linear transfer functions with non-zero gradients, such as $\max(0.1x, x)$.

We state and prove these results more formally in Appendix E.

3. Approximate Gauss-Newton Method

Besides being intractable for large neural networks, the Hessian is not guaranteed to be PSD. A Newton update $H^{-1}g$ could therefore lead to an increase in the error. A common PSD approximation to the Hessian is the Gauss-Newton (GN) matrix. For an error $E(h^L(\theta))$, the sample Hessian is given by:

$$\frac{\partial^2 E}{\partial\theta_i\partial\theta_j} = \sum_k \frac{\partial E}{\partial h_k^L}\frac{\partial^2 h_k^L}{\partial\theta_i\partial\theta_j} + \sum_{k,l}\frac{\partial h_k^L}{\partial\theta_i}\frac{\partial^2 E}{\partial h_k^L\partial h_l^L}\frac{\partial h_l^L}{\partial\theta_j} \quad (11)$$

Assuming that $\mathcal{H}_L$ is PSD, the GN method forms a PSD approximation by neglecting the first term in (11). This can be written in matrix notation as:

$$G \equiv {J_\theta^{h_L}}^\mathsf{T}\mathcal{H}_L J_\theta^{h_L} \quad (12)$$

where $J_\theta^{h_L}$ is the Jacobian of the network outputs with respect to the parameters. The expected GN matrix is the average of (12) over the datapoints:

$$\bar{G} \equiv \mathbb{E}\left[{J_\theta^{h_L}}^\mathsf{T}\mathcal{H}_L J_\theta^{h_L}\right]_{p(x,y)} \quad (13)$$

[5]Note that, for piecewise linear f, E is not necessarily piecewise linear in θ.

[6]This excludes any pathological regions where the objective function has zero curvature.

Whilst (13) shows how to calculate the GN matrix exactly, in practice we cannot feasibly store the matrix in this raw form. To proceed, similar to the Hessian, we will make a block diagonal approximation. As we will show, as for the Hessian itself, even a block diagonal approximation is computationally infeasible, and additional approximations are required. Before embarking on this sequence of approximations, we first show that the GN matrix can be expressed as the expectation of a Khatri-Rao product, *i.e.* blocks of Kronecker products, corresponding to the weights of each layer. We will subsequently approximate the expectation of the Kronecker products as the product of the expectations of the factors, making the blocks efficiently invertible.

3.1. The GN Matrix as a Khatri-Rao Product

Using the definition of $\bar{G}$ in (13) and the chain rule, we can write the block of the matrix corresponding to the parameters in layers λ and β as:

$$\bar{G}_{\lambda,\beta} = \mathbb{E}\left[{J_{W_\lambda}^{h_\lambda}}^\mathsf{T} {J_{h_\lambda}^{h_L}}^\mathsf{T} \mathcal{H}_L J_{h_\beta}^{h_L} J_{W_\beta}^{h_\beta}\right] \tag{14}$$

where $[J_{h_\lambda}^{h_L}]_{i,k} \equiv \frac{\partial h_k^L}{\partial h_i^\lambda}$. Defining $\mathcal{G}_{\lambda,\beta}$ as the pre-activation GN matrix between the λ and β layers' pre-activation vectors:

$$\mathcal{G}_{\lambda,\beta} = {J_{h_\lambda}^{h_L}}^\mathsf{T} \mathcal{H}_L J_{h_\beta}^{h_L} \tag{15}$$

and using the fact that $J_{W_\lambda}^{h_\lambda} = a_{\lambda-1}^\mathsf{T} \otimes I$ we obtain

$$\bar{G}_{\lambda,\beta} = \mathbb{E}\left[\left(a_{\lambda-1} a_{\beta-1}^\mathsf{T}\right) \otimes \mathcal{G}_{\lambda,\beta}\right] \tag{16}$$

We can therefore write the GN matrix as the expectation of the Khatri-Rao product:

$$\bar{G} = \mathbb{E}\left[\mathcal{Q} \star \mathcal{G}\right] \tag{17}$$

where the blocks of $\mathcal{G}$ are the pre-activation GN matrices $\mathcal{G}_{\lambda,\beta}$ as defined in (16), and the blocks of $\mathcal{Q}$ are:

$$\mathcal{Q}_{\lambda,\beta} \equiv a_{\lambda-1} a_{\beta-1}^\mathsf{T} \tag{18}$$

3.2. Approximating the GN Diagonal Blocks

For simplicity, from here on we denote by G_λ the diagonal blocks of the *sample* GN matrix with respect to the weights of layer λ (dropping the duplicate index). Similarly, we drop the index for the diagonal blocks $\mathcal{Q}_\lambda$ and $\mathcal{G}_\lambda$ of the corresponding matrices in (17), giving more compactly:

$$G_\lambda = \mathcal{Q}_\lambda \otimes \mathcal{G}_\lambda \tag{19}$$

The diagonal blocks of the expected GN $\bar{G}_\lambda$ are then given by $\mathbb{E}\left[G_\lambda\right]$. Computing this requires evaluating a block diagonal matrix for each datapoint and accumulating the result. However, since the expectation of a Kronecker product is not necessarily Kronecker factored, one would need to explicitly store the whole matrix $\bar{G}_\lambda$ to perform this accumulation. With D being the dimensionality of a layer, this matrix would have $O(D^4)$ elements. For D of the order of 1000, it would require several terabytes of memory to store $\bar{G}_\lambda$. As this is prohibitively large, we seek an approximation for the diagonal blocks that is both efficient to compute and store. The approach we take is the factorised approximation:

$$\mathbb{E}\left[G_\lambda\right] \approx \mathbb{E}\left[\mathcal{Q}_\lambda\right] \otimes \mathbb{E}\left[\mathcal{G}_\lambda\right] \tag{20}$$

Under this factorisation, the updates for each layer can be computed efficiently by solving a Kronecker product form linear system – see the supplementary material. The first factor $\mathbb{E}\left[\mathcal{Q}_\lambda\right]$ is simply the uncentered covariance of the activations:

$$\mathbb{E}\left[\mathcal{Q}_\lambda\right] = \frac{1}{N} A_{\lambda-1} A_{\lambda-1}^\mathsf{T} \tag{21}$$

where the n^{th} column of the $d \times n$ matrix $A_{\lambda-1}$ is the set of activations of layer $\lambda-1$ for datapoint n. The second factor $\mathbb{E}\left[\mathcal{G}_\lambda\right]$, can be computed efficiently, as described below.

3.3. The Pre-Activation Recursion

Analogously to the block diagonal pre-activation Hessian recursion (8), a similar recursion can be derived for the pre-activation GN matrix diagonal blocks:

$$\mathcal{G}_\lambda = B_\lambda W_{\lambda+1}^\mathsf{T} \mathcal{G}_{\lambda+1} W_{\lambda+1} B_\lambda \tag{22}$$

where the recursion is initialised with the Hessian of the output $\mathcal{H}_L$.

This highlights the close relationship between the pre-activation Hessian recursion and the pre-activation GN recursion. Inspecting (8) and (22) we notice that the only difference in the recursion stems from terms containing the diagonal matrices D_λ. From (7) and (16) it follows that in the case of piecewise linear transfer functions, the diagonal blocks of the Hessian are equal to the diagonal blocks of the GN matrix[7].

Whilst this shows how to calculate the sample pre-activation GN blocks efficiently, from (20) we require the calculation of the *expected* blocks $\mathbb{E}\left[\mathcal{G}_\lambda\right]$. In principle, the recursion could be applied for every data point. However, this is impractical in terms of the computation time and a vectorised implementation would impose infeasible memory requirements. Below, we show that when the number of outputs is small, it is in fact possible to efficiently compute the exact expected pre-activation GN matrix diagonals. For the case of a large number of outputs, we describe a further approximation to $\mathbb{E}\left[\mathcal{G}_\lambda\right]$ in Section 3.5.

[7] This holds only at points where the derivative exists.

3.4. Exact Low Rank Calculation of $\mathbb{E}[\mathcal{G}_\lambda]$

Many problems in classification and regression deal with a relatively small number of outputs. This implies that the rank K of the output layer GN matrix $\mathcal{G}_L$ is low. We use the square root representation:

$$\mathcal{G}_\lambda = \sum_{k=1}^{K} C_\lambda^k C_\lambda^{k\mathsf{T}} \tag{23}$$

From (22) we then obtain the recursion:

$$C_\lambda^k = B_\lambda W_{\lambda+1}^\mathsf{T} C_{\lambda+1}^k \tag{24}$$

This allows us to calculate the expectation as:

$$\mathbb{E}[\mathcal{G}_\lambda] = \mathbb{E}\left[\sum_k C_\lambda^k C_\lambda^{k\mathsf{T}}\right] = \frac{1}{N}\sum_k \tilde{C}_\lambda^k \left(\tilde{C}_\lambda^k\right)^\mathsf{T} \tag{25}$$

where we stack the column vectors C_λ^k for each datapoint into a matrix $\tilde{C}_k^\lambda$, analogous to (21). Since we need to store only the vectors C_λ^k per datapoint, this reduces the memory requirement to $K \times D \times N$; for small K this is a computationally viable option. We call this method Kronecker Factored Low Rank (KFLR).

3.5. Recursive Approximation of $\mathbb{E}[\mathcal{G}_\lambda]$

For higher dimensional outputs, *e.g.* in autoencoders, rather than backpropagating a sample pre-activation GN matrix for every datapoint, we propose to simply pass the expected matrix through the network. This yields the nested expectation approximation of (22):

$$\mathbb{E}[\mathcal{G}_\lambda] \approx \mathbb{E}\left[B_\lambda W_{\lambda+1}^\mathsf{T}\, \mathbb{E}[\mathcal{G}_{\lambda+1}]\, W_{\lambda+1} B_\lambda\right] \tag{26}$$

The recursion is initialised with the exact value $\mathbb{E}[\mathcal{G}_L]$. The method will be referred to as Kronecker Factored Recursive Approximation (KFRA).

4. Related Work

Despite the prevalence of first-order methods for neural network optimisation, there has been considerable recent interest in developing practical second-order methods, which we briefly outline below.

Martens (2010) and Martens & Sutskever (2011) exploited the fact that full Gauss-Newton matrix-vector products can be computed efficiently using a form of automatic differentiation. This was used to approximately solve the linear system $\bar{G}\delta = \nabla f$ using conjugate gradients to find the parameter update δ. Despite making good progress on a per-iteration basis, having to run a conjugate gradient descent optimisation at every iteration proved too slow to compete with well-tuned first-order methods.

The closest related work to that presented here is the KFAC method (Martens & Grosse, 2015), in which the Fisher matrix is used as the curvature matrix. This is based on the output y of the network defining a conditional distribution $p_\theta(y|x)$ on the observation y, with a loss defined as the KL-divergence between the empirical distribution $p(y|x)$ and the network output distribution. The network weights are chosen to minimise the KL-divergence between the conditional output distribution and the data distribution. For example, defining the network output as the mean of a fixed variance Gaussian or a Bernoulli/Categorical distribution yields the common squared error and cross-entropy objectives respectively.

Analogously to our work, Martens & Grosse (2015) develop a block-diagonal approximation to the Fisher matrix. The Fisher matrix is another PSD approximation to the Hessian that is used in natural gradient descent (Amari, 1998). In general, the Fisher and GN matrices are different. However, for the case of $p_\theta(y|x)$ defining an exponential family distribution, the Fisher and GN matrices are equivalent, see Appendix C.3. As in our work, Martens & Grosse (2015) use a factorised approximation of the form (20). However, they subsequently approximate the expected Fisher blocks by drawing Monte Carlo samples of the gradients from the conditional distribution defined by the neural network. As a result, KFAC is always an approximation to the GN pre-activation matrix, whereas our method can provide an exact calculation of $\mathbb{E}[\mathcal{G}]$ in the low rank setting. See also Appendix C.4 for differences between our KFRA approximation and KFAC.

More generally, our method does not require any probabilistic model interpretation and is therefore more widely applicable than KFAC.

5. Experiments

We performed experiments[8] training deep autoencoders on three standard grey-scale image datasets and classifying hand-written digits as odd or even. The datasets are:

MNIST consists of $60,000$ 28×28 images of hand-written digits. We used only the first $50,000$ images for training (since the remaining $10,000$ are usually used for validation).

CURVES contains $20,000$ training images of size 28×28 pixels of simulated hand-drawn curves, created by choosing three random points in the 28×28 pixel plane (see the supplementary material of (Hinton & Salakhutdinov, 2006) for details).

[8]Experiments were run on a workstation with a Titan Xp GPU and an Intel Xeon CPU E5-2620 v4 @ 2.10GHz.

FACES is an augmented version of the Olivetti faces dataset (Samaria & Harter, 1994) with 10 different images of 40 people. We follow (Hinton & Salakhutdinov, 2006) in creating a training set of $103,500$ images by choosing 414 random pairs of rotation angles (-90 to 90 degrees) and scaling factors (1.4 to 1.8) for each of the 250 images for the first 25 people and then subsampling to 25×25 pixels.

We tested the performance of second-order against first-order methods and compared the quality of the different GN approximations. In all experiments we report only the training error, as we are interested in the performance of the optimiser rather than how the models generalise.

When using second-order methods, it is important in practice to adjust the unmodified update δ in order to dampen potentially over-confident updates. One of our central interests is to compare our approach against KFAC. We therefore followed (Martens & Grosse, 2015) as closely as possible, introducing damping in an analogous way. Details on the implementation are in Appendix B. We emphasise that throughout all experiments we used the default damping parameter settings, with no tweaking required to obtain acceptable performance[9].

Additionally, as a form of momentum for the second-order methods, we compared the use of a moving average with a factor of 0.9 on the curvature matrices $\mathcal{G}_\lambda$ and $\mathcal{Q}_\lambda$ to only estimating them from the current minibatch. We did not find any benefit in using momentum on the updates themselves; on the contrary this made the optimisation unstable and required clipping the updates. We therefore do not include momentum on the updates in our results.

All of the autoencoder architectures are inspired by (Hinton & Salakhutdinov, 2006). The layer sizes are D-1000-500-250-30-250-500-1000-D, where D is the dimensionality of the input. The grey-scale values are interpreted as the mean parameter of a Bernoulli distribution and the loss is the binary cross-entropy on CURVES and MNIST, and square error on FACES.

5.1. Comparison to First-Order Methods

We investigated the performance of both KFRA and KFAC compared to popular first-order methods. Four of the most prevalent gradient-based optimisers were considered – Stochastic Gradient Descent, Nesterov Accelerated Gradient, Momentum and ADAM (Kingma & Ba, 2014). A common practice when using first-order methods is to decrease the learning rate throughout the training procedure. For this reason we included an extra parameter T – the decay period – to each of the methods, halving the learning rate every T iterations. To find the best first-order method, we ran a grid search over these two hyperarameters[10].

Each first-order method was run for $40,000$ parameter updates for MNIST and CURVES and $160,000$ updates for FACES. This resulted in a total of 35 experiments and $1.4/5.6$ million updates for each dataset per method. In contrast, the second-order methods did not require adjustment of any hyperparameters and were run for only $5,000/20,000$ updates, as they converged much faster[11]. For the first-order methods we found ADAM to outperform the others across the board and we consequently compared the second-order methods against ADAM only.

Figure 2 shows the performance of the different optimisers on all three datasets. We present progress both per parameter update, to demonstrate that the second-order optimisers effectively use the available curvature information, and per GPU wall clock time, as this is relevant when training a network in practice. For ADAM, we display the performance using the default learning rate 10^{-3} as well as the top performing combination of learning rate and decay period. To illustrate the sensitivity of ADAM to these hyperparameter settings (and how much can therefore be gained by parameter tuning) we also plot the average performance resulting from using the top 10 and top 20 settings.

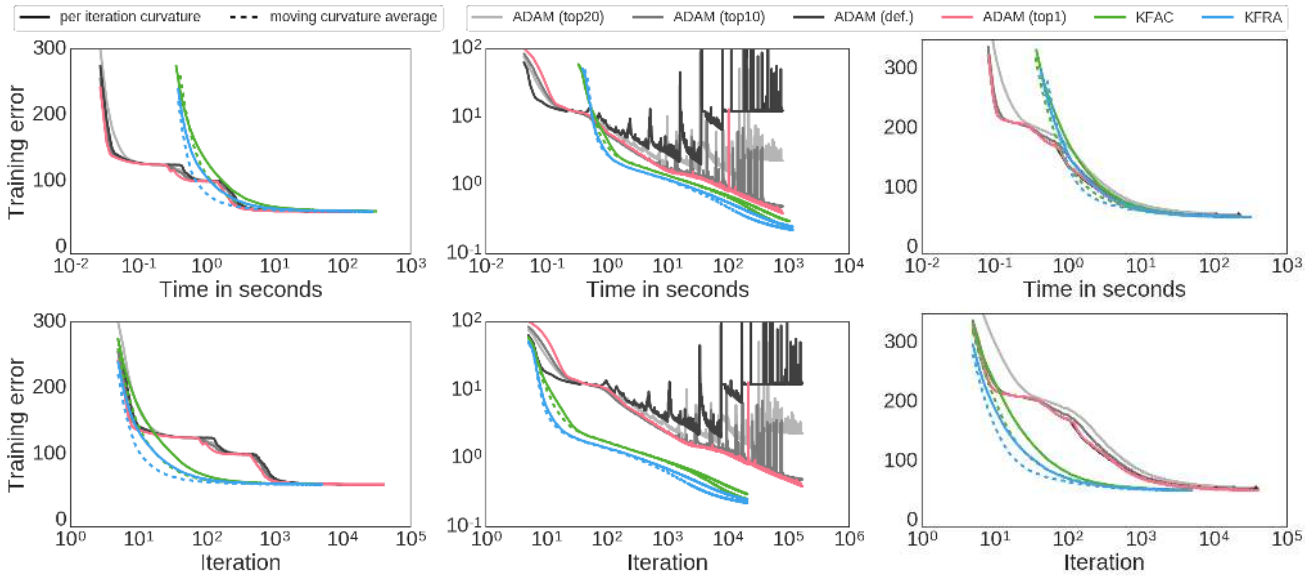
Even after significantly tuning the ADAM learning rate and decay period, the second-order optimisers outperformed ADAM out-of-the-box across all three datasets. In particular on the challenging FACES dataset, the optimisation was not only much faster when using second-order methods, but also more stable. On this dataset, ADAM appears to be highly sensitive to the learning rate and in fact diverged when run with the default learning rate of 10^{-3}. In contrast to ADAM, the second-order optimisers did not get trapped in plateaus in which the error does not change significantly.

In comparison to KFAC, KFRA showed a noticeable speed-up in the optimisation both per-iteration and when measuring the wall clock time. Their computational cost for each update is equivalent in practice, which we discuss in detail in Appendix C.4. Thus, to validate that the advantage of KFRA over KFAC stems from the quality of its updates, we compared the alignment of the updates of each method with the exact Gauss-Newton update (using the slower Hessian-free approach; see Appendix F.2 for the figures). We found that KFRA tends to be more closely aligned with the exact Gauss-Newton update, which provides a possible explana-

[9] Our damping parameters could be compared to the exponential decay parameters β_1 and β_2 in ADAM, which are typically left at their recommended default values.

[10] We varied the learning rate from 2^{-6} to 2^{-13} at every power of 2 and chose the decay period as one of $\{100\%, 50\%, 25\%, 12.5\%, 6.25\%\}$ of the number of updates.

[11] For fair comparison, all of the methods were implemented using Theano (Theano Development Team, 2016) and Lasagne (Dieleman et al., 2015).



(a) CURVES (b) FACES (c) MNIST

Figure 2. Comparison of the objective function being optimised by KFRA, KFAC and ADAM on CURVES, FACES and MNIST. GPU benchmarks are in the first row, progress per update in the second. The dashed line indicates the use of momentum on the curvature matrix for the second-order methods. Errors are averaged using a sliding window of ten.

tion for its better performance.

5.2. Non-Exponential Family Model

To compare our approximate Gauss-Newton method and KFAC in a setting where the Fisher and Gauss-Newton matrix are not equivalent, we designed an experiment in which the model distribution over y is not in the exponential family. The model is a mixture of two binary classifiers[12]:

$$p(y|h_L) = \sigma(h_1^L)\sigma(h_2^L)^y\sigma(-h_2^L)^{1-y} + (1-\sigma(h_1^L))\sigma(h_3^L)^y\sigma(-h_3^L)^{1-y} \tag{27}$$

We used the same architecture as for the encoding layers of the autoencoders – D-1000-500-250-30-1, where $D = 784$ is the size of the input. The task of the experiment was to classify MNIST digits as even or odd. Our choice was motivated by recent interest in neural network mixture models (Eigen et al., 2013; Zen & Senior, 2014; van den Oord & Schrauwen, 2014; Shazeer et al., 2017); our mixture model is also appropriate for testing the performance of KFLR. Training was run for $40,000$ updates for ADAM with a grid search as in Section 5.1, and for $5,000$ updates for the second-order methods. The results are shown in Figure 3.

For the CPU, both per iteration and wall clock time the second-order methods were faster than ADAM; on the GPU, however, ADAM was faster per wall clock time. The value of the objective function at the final parameter values was higher for second-order methods than for ADAM. However, it is important to keep in mind that all methods achieved a nearly perfect cross-entropy loss of around 10^{-8}. When so close to the minimum we expect the gradients and curvature to be very small and potentially dominated by noise introduced from the mini-batch sampling. Additionally, since the second-order methods invert the curvature, they are more prone to accumulating numerical errors than first-order methods, which may explain this behaviour close to a minimum.

Interestingly, KFAC performed almost identically to KFLR, despite the fact that KFLR computes the exact pre-activation Gauss-Newton matrix. This suggests that in the low-dimensional output setting, the benefits from using the exact low-rank calculation are diminished by the noise and the rather coarse factorised Kronecker approximation.

6. Rank of the Empirical Curvature

The empirical success of second-order methods raises questions about the curvature of the error function of a neural network. As we show in Appendix D the Monte Carlo Gauss-Newton matrix rank is upper bounded by the rank of the last layer Hessian times the size of the mini-batch. More generally, the rank is upper bounded by the rank of $\mathcal{H}_L$ times the size of the data set. As modern neural networks commonly have millions of parameters, the exact Gauss-Newton matrix is usually severely underdetermined. This implies that the curvature will be zero in many directions. This phenomenon is particularly pro-

[12] In this context $\sigma(x) = (1 + \exp(-x))^{-1}$.

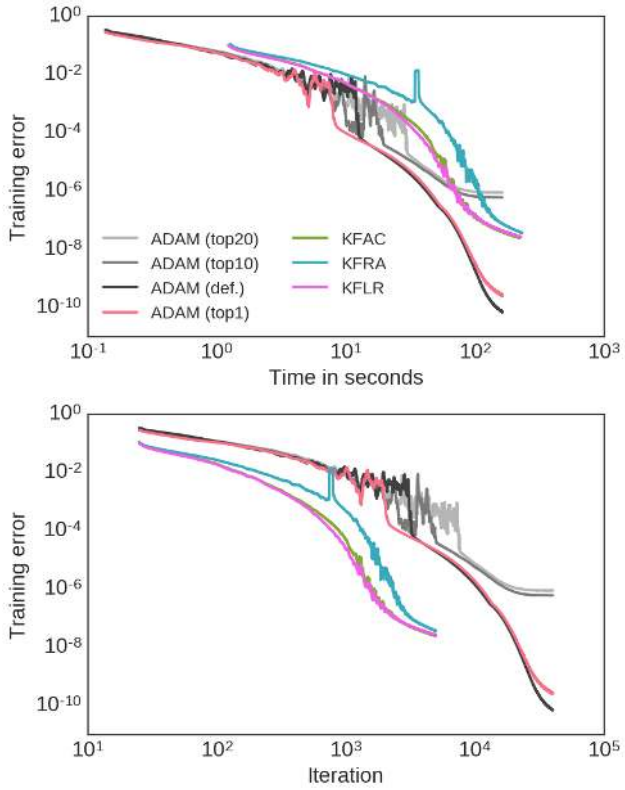


Figure 3. Comparative optimisation performance on an MNIST binary mixture-classification model. We used momentum on the curvature matrix for all methods, as it stabilises convergence.

nounced for the binary classifier in Section 5.2, where the rank of the output layer Hessian is one.

We can draw a parallel between the curvature being zero and standard techniques where the maximum likelihood problem is under-determined for small data sets. This explains why damping is so important in such situations, and its role goes beyond simply improving the numerical stability of the algorithm. Our results suggest that, whilst in practice the Gauss-Newton matrix provides curvature only in a limited parameter subspace, this still provides enough information to allow for relatively large parameter updates compared to gradient descent, see Figure 2.

7. Conclusion

We presented a derivation of the block-diagonal structure of the Hessian matrix arising in feedforward neural networks. This leads directly to the interesting conclusion that for networks with piecewise linear transfer functions and convex loss the objective has no differentiable local maxima. Furthermore, with respect to the parameters of a single layer, the objective has no differentiable saddle points. This may provide some partial insight into the success of such transfer functions in practice.

Since the Hessian is not guaranteed to be positive semi-definite, two common alternative curvature measures are the Fisher matrix and the Gauss-Newton matrix. Unfortunately, both are computationally infeasible and, similar to Martens & Grosse (2015), we therefore used a block diagonal approximation, followed by a factorised Kronecker approximation. Despite parallels with the Fisher approach, formally the two methods are different. Only in the special case of exponential family models are the Fisher and Gauss-Newton matrices equivalent; however, even for this case, the subsequent approximations used in the Fisher approach (Martens & Grosse, 2015) differ from ours. Indeed, we showed that for problems in which the network has a small number of outputs no additional approximations are required. Even on models where the Fisher and Gauss-Newton matrices are equivalent, our experimental results suggest that our KFRA approximation performs marginally better than KFAC. As we demonstrated, this is possibly due to the updates of KFRA being more closely aligned with the exact Gauss-Newton updates than those of KFAC.

Over the past decade first-order methods have been predominant for Deep Learning. Second-order methods, such as Gauss-Newton, have largely been dismissed because of their seemingly prohibitive computational cost and potential instability introduced by using mini-batches. Our results on comparing both the Fisher and Gauss-Newton approximate methods, in line with (Martens & Grosse, 2015), confirm that second-order methods can perform admirably against even well-tuned state-of-the-art first-order approaches, while not requiring any hyperparameter tuning.

In terms of wall clock time on a CPU, in our experiments, the second-order approaches converged to the minimum significantly more quickly than state-of-the-art first-order methods. When training on a GPU (as is common in practice), we also found that second-order methods can perform well, although the improvement over first-order methods was more marginal. However, since second-order methods are much faster per update, there is the potential to further improve their practical utility by speeding up the most expensive computations, specifically solving linear systems on parallel compute devices.

Acknowledgements

We thank the reviewers for their valuable feedback and suggestions. We also thank Raza Habib, Harshil Shah and James Townsend for their feedback on earlier drafts of this paper. Finally, we are grateful to James Martens for helpful discussions on the implementation of KFAC.

References

Amari, S.-I. Natural Gradient Works Efficiently in Learning. *Neural Computation*, 10(2):251–276, 1998.

Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in Neural Information Processing Systems*, pp. 2933–2941, 2014.

Dieleman, S., Schlüter, J., Raffel, C., Olson, E., Sønderby, S. K., Nouri, D., et al. Lasagne: First Release, August 2015.

Duchi, J., Hazan, E., and Singer, Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *The Journal of Machine Learning Research*, 12: 2121–2159, 2011.

Eigen, D., Ranzato, M., and Sutskever, I. Learning Factored Representations in a Deep Mixture of Experts. *arXiv preprint arXiv:1312.4314*, 2013.

Gower, R. M. and Gower, A. L. Higher-Order Reverse Automatic Differentiation with Emphasis on the Third-Order. *Mathematical Programming*, 155(1-2):81–103, 2016.

Hinton, G. E. and Salakhutdinov, R. R. Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507, 2006.

Kingma, D. and Ba, J. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Martens, J. Deep Learning via Hessian-Free Optimization. In *Proceedings of the 27th International Conference on Machine Learning*, pp. 735–742, 2010.

Martens, J. New Insights and Perspectives on the Natural Gradient Method. *arXiv preprint arXiv:1412.1193*, 2014.

Martens, J. and Grosse, R. B. Optimizing Neural Networks with Kronecker-factored Approximate Curvature. In *Proceedings of the 32nd International Conference on Machine Learning*, pp. 2408–2417, 2015.

Martens, J. and Sutskever, I. Learning Recurrent Neural Networks with Hessian-Free Optimization. In *Proceedings of the 28th International Conference on Machine Learning*, pp. 1033–1040, 2011.

Nesterov, Y. A Method of Solving a Convex Programming Problem with Convergence Rate O (1/k2). *Soviet Mathematics Doklady*, 27(2):372–376, 1983.

Pearlmutter, B. A. Fast Exact Multiplication by the Hessian. *Neural Computation*, 6(1):147–160, 1994.

Polyak, B. T. Some Methods of Speeding up the Convergence of Iteration Methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

Samaria, F. S. and Harter, A. C. Parameterisation of a Stochastic Model for Human Face Identification. In *Proceedings of the Second IEEE Workshop on Applications of Computer Vision*, pp. 138–142. IEEE, 1994.

Schaul, T., Zhang, S., and LeCun, Y. No More Pesky Learning Rates. In *Proceedings of the 30th International Conference on Machine Learning*, pp. 343–351, 2013.

Schraudolph, N. N. Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent. *Neural Computation*, 14(7):1723–1738, 2002.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. *arXiv preprint arXiv:1701.06538*, 2017.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. On the Importance of Initialization and Momentum in Deep Learning. In *Proceedings of the 30th International Conference on Machine Learning*, pp. 1139–1147, 2013.

Theano Development Team. Theano: A Python Framework for Fast Computation of Mathematical Expressions. *arXiv preprint arXiv:1605.02688*, 2016.

van den Oord, A. and Schrauwen, B. Factoring Variations in Natural Images with Deep Gaussian Mixture Models. In *Advances in Neural Information Processing Systems*, pp. 3518–3526, 2014.

Zeiler, M. D. Adadelta: An Adaptive Learning Rate Method. *arXiv preprint arXiv:1212.5701*, 2012.

Zen, H. and Senior, A. Deep Mixture Density Networks for Acoustic Modeling in Statistical Parametric Speech Synthesis. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 3844–3848. IEEE, 2014.