Second-order optimisation methods in Deep Learning with applications on the Fashion MNIST dataset

Ben Veitch [*] [1]

Abstract

I present an empirical study into the use of second-order solvers, L-BFGS and NLCG for training neural networks. Comparisons are made on the Fashion MNIST dataset and I show improved performance in terms of convergence and dev-accuracy over Gradient descent. Whilst L-BFGS and NLCG have a higher cost per iteration than gradient descent, being roughly twice that of gradient descent, the improvement in total training time might more than compensate for the increased iteration cost. I show that the use of stochastic sampling which is required for efficient training on a large dataset does harm the convergence of these solvers. As such, batches must be fixed throughout the training epochs. I also present a sufficient condition, Theorem 4.1, for convexity of the loss-function during training. I argue that this can incorporated into a Gauss-Newton scheme to determine effective preconditioning strategies for improving convergence during training.

1. Introduction

Large-scale optimisation methods are fundamental to machine learning. This is due to the ubiquity of large datasets in the modern world and the nature of the numerical methods which are used to train a neural network. For example, the Fashion-MNIST dataset (Zalando Research, 2017) consists of 60,000 training images of 10 clothing types with 28 $\times$ 28 pixels in each image, this is around 470M parameters. To train an accurate model to adequate performance on a large dataset can require neural networks with many layers; LeNet-5 (LeCun et al., 1998) had 7 layers, whereas later versions of Resnet (He et al., 2016) have 158, and numbers in excess of 1200 are reported as being possible (Huang et al., 2016). This leads to many millions of parameters for the neural network weights all of which must be optimized for. The development of algorithms which are scalable with respect to data size whilst also being fast to converge is therefore key to the success of deep learning methods.

The ability to quickly train sets of neural network weights has the advantage of reducing the cost of cross-validation of hyper parameters which leads to an improvement in the test accuracy (low bias, low variance) of deep neural networks. When neural networks are slow to converge this leads to long turnaround time and additional confusion can be introduced by extracting the weights before the solver has finished converging. A high-variance solution may be avoided at the expense of higher bias, whereas in this case neither of these are legitimately associated with the hyper-parameters of the network itself.

However, the challenge associated with training very deep networks is not simply one of efficiency. Very often a first-order method is used, (using only the local gradient) and this leads to significant theoretical problems. A well publicised example is the problem of 'vanishing-exploding gradients' (see Huang et al., 2016). Here, the weights associated to a particular layer can become vanishingly small or exponentially large, both of these greatly reduce the effective size of the weights in that layer whilst also introducing numerical errors. This leads to a loss of predictive power for the network. A second problem is the existence of saddle points on the surface of the loss function (see Martens, 2010). These pose a difficulty for any solver but are particularly pathological if only the local gradient is considered. In this case, the loss function will likely become trapped in a local minimum and will also lose predictive power. A final problem is the determination of an accurate step-length. Clearly, when the step-length is poorly estimated, even for a convex function with an accurate local gradient the resulting weights update can be a long way from the true minimum and convergence will be slow.

In contrast to gradient descent, second-order numerical optimization methods such as Newton's method, non-linear conjugate gradients (NLCG) or BFGS can converge quickly (see Nocedal and Wright, 2006). They owe much of this fast convergence to capturing some of the local curvature of the optimization surface and therefore avoid problems associated to steep valleys, corners and saddle points which are typically present in the minimum surface of a nonlinear objective function. However, the ability to capture local curvature requires an approximate Hessian and this can become computationally expensive for models with millions of parameters. The advantage of NLCG and BFGS is that they can efficiently approximate the action of the Hessian without incurring the full cost of a Newton scheme.

On the other hand stochastic descent methods which draw random samples, or mini-batches, from the training set scale well with data size but can converge slowly. See Goodfellow et al. (2016), Hinton (2017) or Ruder (2017) for clear introductions to these methods. As the data is drawn from small batches recovering the local curvature may become unstable. Furthermore, finding an accurate step-length becomes problematic and more ad-hoc methods are required to improve convergence. These include but are not limited to Momentum (Qian, 1999), Adagrad (Duchi et al., 2011), and Adam (Kingma and Lei Ba, 2015).

From these observations it is easy to argue for the inclusion of second-order schemes together with batch sampling to improve convergence when training neural networks. This paper is not the first to make this proposal, and there is a significant literature on this approach. Some examples of recent papers are; Asi and Duchi (2019), Le et al. (2011), Martens (2010), Xu et al. (2018) and Zhao et al. (2018).

Le et al. (2011), show that Limited memory BFGS (L-BFGS) and NLCG can significantly speed up and simplify the training of deep algorithms when compared to conventional stochastic gradient descent. The authors claim that L-BFGS is superior for low-dimensional convolutional models, whereas for high dimensional problems NLCG is most competitive. Amongst the algorithms investigated are Restricted Boltzman Machines, auto-encoders, and Sparse RBMs. The authors also discuss the Map-Reduce framework for enhancing parallel computation when training convolutional neural networks CNNs. For this architecture L-BFGS had the best performance. Results were obtained for training on the standard MNIST dataset. The authors also used this dataset to confirm that the use of L-BFGS did not adversely affect classification accuracy. However, Le et al. do not discuss stochastic sampling within second-order schemes and how choices here might impact performance. To an extent the use of parallel computation might mitigate the need for stochastic sampling but this does not resolve the theoretical challenge posed by large-scale data.

Martens (2010) looks at second-order optimisation methods for training deep auto-encoders. He achieves superior results to those using first-order methods with pre-training. Martens is in agreement with the present paper that many of the difficulties experienced when training deep networks stem from the choice of first-order methods. Martens mainly focusses on the conjugate-gradient methodology. The paper describes the importance of using fixed batches when combining mini-batch sampling with CG methods and there is a discussion of termination criterion for CG schemes within his Hessian-free (truncated Gauss-Newton) approach. Martens' work is also noticeable for using a preconditioner within the conjugate gradient algorithm. A diagonal Gauss-Newton matrix is used for this which is summed over all samples within each mini-batch. As was found by Le et al. (2011), performance was best using relatively large mini-batches. No doubt this helps reduce the influence of outliers within small batches and improves the local curvature estimate.

Xu et al. (2018) also present an empirical study of second-order schemes. In this paper, trust region (TR) and adaptive regularization with cubics (ARC) methods are investigated. The authors show that these methods are competitive with SGD using Momentum but have the additional benefit of being highly robust to hyper-parameter settings. Experiments are performed on the CIFAR-10 dataset using shallow networks. Xu et al. make comparisons of TR and ARC with L-BFGS showing that TR and ARC have improved robustness over L-BFGS with respect to the choice of initialization method. These authors present six criteria to assess second-order optimization in deep learning; *computational efficiency*, *robustness to hyper-parameters*, *escaping saddle-points*, *generalization performance*, *benefits of sub-sampling*, and *comparison among second-order methods*.

The paper of Asi and Duchi (2019), is more theoretical and is aimed at improving the robustness of stochastic optimization methods to hyper-parameter choices. The authors define a concept of stability for stochastic solvers and discuss its importance in light of convex optimization theory and proximal methods (see Boyd and Vandenberghe, 2004). The main concern of these authors is improving the sensitivity (stability) of stochastic optimization schemes with respect to the choice of an initial step-length. They are successful in this regard and show that stochastic schemes offer improved convergence for a larger range of initial step-lengths. The authors also compare results with ADAM showing that ADAM can converge faster to a tolerance level but has a narrower range of step-lengths for which it is convergent. Comparisons were made on CIFAR10 (similar to Fashion MNIST) and the Stanford Dogs dataset. The authors use the same approximate Hessian when training Neural networks as was used as a preconditioner by Martens (2010). Asi and Duchi (2019) are critical of the number of computer and engineer hours which can be wasted by picking hyper-parameters and repeatedly running cross-validation tests on deep networks which can take weeks to train on a single model. They estimate that the energy expended during training of a recurrent deep network (Collins et al., 2016) is, 'sufficient to drive 4,000 Toyota Camrys the 380 miles between San Francisco and Los Angeles'

The paper of Zhao et al. (2018) is a mathematically sophisticated work aimed at comparing stochastic L-BFGS methods to its competitors. The authors prove that a stochastic L-BFGS solver will converge linearly in expectation and show that the number of outer iterations to achieve a tolerance of ϵ is $O(\log(1/\epsilon))$. An algorithm is also provided for implementing stochastic L-BFGS. The authors provide the results of experiments training logistic regression with Stochastic L-BFGS and these were in agreement with the $O(\log(1/\epsilon))$ sub-optimality bound. This is a significant improvement on the $O(1/\epsilon^2)$ usually cited for stochastic gradient descent (see Srebro and Tewari, 2010).

The modest aim of this paper is to make empirical comparisons between gradient descent, L-BFGS and NLCG schemes by experimenting with training on the Fashion MNIST dataset. My focus will be on the interaction between the choice of stochastic sampling method and the performance of the solver. I also include an investigation into how a Hessian can be computed for a Neural network. Both NLCG and L-BFGS-B solvers are available in scipy.optimize (see SciPy Team, 2020, SciPy v1.7.1) and I shall use these for training small networks. I will not use stochastic implementations of these solvers such as those proposed by Zhao et al. (2018) or Hong and Wilford (2013). The key hypotheses of this work are,

- Including second-order derivatives through the use of an approximate Hessian or using a solver such as NLCG or BFGS *together with* stochastic sampling of mini-batches improves convergence over mini-batch gradient descent alone.

- Including second-order derivatives in the manner of 1 helps mitigate difficulties such as ‘vanishing/exploding gradients’ and poor line-search estimation.

2. Second-order optimization algorithms

Numerical optimisation (see Nocedal and Wright, 2006) has a long history dating back to Newton and Gauss. In general, given a loss function $\mathcal{L}$ expressed as a function of a model m, with a gradient $\nabla\mathcal{L}$ then Newton's descent step takes the form,

$$m_{(k+1)} = m_{(k)} - \alpha_k H_{(k)}^{-1} \nabla\mathcal{L}(m_{(k)}), \tag{1}$$

where α_k is the current step-length and H[1] is the symmetric positive-definite Hessian matrix,

$$H_{ij} = \frac{\partial^2 \mathcal{L}}{\partial m_i \partial m_j}. \tag{2}$$

Two issues are immediately apparent, first for large scale nonlinear problems the calculation of H is not straightforward and efficiently calculating H^{-1} raises serious numerical challenges. Both of these problems are encountered when training weights in deep neural networks. For Fashion MNIST, using a single hidden layer the weight matrices had sizes of 784 x 300 for hidden layer and 300 x 10 for output layer with biases taking sizes 300 and 10 leading to a total size (for the gradient) of 238,510 or 1.82 Mb. The Hessian therefore is 238,510 x 238,510 or 3.31 Mb.

Algorithms such as nonlinear conjugate gradients (NLCG) and BFGS avoid solving for the inverse Hessian directly and instead solve the equivalent convex-quadratic minimization problem,

$$j(m) = \frac{1}{2} m^T H m - m^T \nabla\mathcal{L}. \tag{3}$$

In outline these algorithms perform the iterative procedure,

$$m'_{(k)} = \operatorname{argmin} j(m_{(k)}), \tag{4}$$

$$m_{(k+1)} = m_{(k)} + \alpha_k m'_{(k)}. \tag{5}$$

In the case of conjugate gradients the minimization is solved iteratively by decomposing the model descent into a set of residual and conjugate directions. I don't care to review the full details here, Shewchuk (1994) is an excellent tutorial on both the mathematics and implementation details.

An alternative to NLCG is BFGS which attempts to solve the same minimum problem but builds up approximations to the Hessian by calculating finite-difference derivatives of the gradient. As the Hessian is built up numerically as opposed to relying on an approximate Hessian this scheme has certain convergence advantages over NLCG which uses an inexact Hessian. L-BFGS is a limited memory implementation of BFGS which avoids storing many past iterations of the approximate Hessian.

It is a matter of custom to illustrate the fast convergence of second-order schemes over first-order using the Rosenbrock function. The Rosenbrock function is defined by,

$$\text{Rosenb}(x, y) = (a - x)^2 + b(y - x^2)^2, \tag{6}$$

with a unique minimum at $(1, 1)$. In Figure 1 I compare the (x, y) iterations in the 2D plane for i. gradient descent, ii. Newton's method and iii. gradient decent with a line-search estimation. Convergence plots of the objective function against iteration are shown in Figure 2. The superior (quadratic convergence) of Newton's scheme is clear from Figure 2 whilst gradient descent has difficulty navigating the curved ravine, in Figure 1, especially without a line-search.

2.1. Preconditioned second-order schemes

Whilst the computation of H is often intractable, it is often the case that it can be approximated by the positive-semi definite form,

$$H \approx P^T \widehat{H} P, \tag{7}$$

[1] Note that when H is the identity matrix this is gradient descent.

where $\widehat{H}$ is computational tractable, for example being diagonal or block-diagonal. Now, if we define the constrained objective function,

$$\widehat{\mathcal{L}}(p) = \mathcal{L}(m), \quad p = Pm \tag{8}$$

then,

$$P^T \nabla \widehat{\mathcal{L}} = \nabla \mathcal{L}, \tag{9}$$

$$P^T \nabla^2 \widehat{\mathcal{L}} P = \nabla^2 \mathcal{L}, \tag{10}$$

and

$$\widehat{j}(p) = \frac{1}{2} p^T \widehat{H} p - p^T \nabla \widehat{\mathcal{L}}. \tag{11}$$

This leads to an efficient second-order scheme,

$$p'_{(k)} = \operatorname{argmin} \widehat{j}(Pm_{(k)}), \tag{12}$$

$$m_{(k+1)} = m_{(k)} + \alpha_k P^{-1} p'_{(k)}. \tag{13}$$

Given a Hessian H a suitable P and $\widehat{H}$ can be found by (block) Cholesky decomposition or (block) LDL decomposition. However, as this is not always practical, it is sufficient to produce an invertible P and a heuristic $\widehat{H}$ which captures salient features of the true Hessian. To this end, Asi and Duchi (2019) and Martens (2010) use the approximate Gauss-Newton Hessian,

$$H_k = \operatorname{Diag}\left(\sum_{i=1}^{k} \nabla \mathcal{L}_i \nabla \mathcal{L}_i^T\right)^{\frac{1}{2}}, \tag{14}$$

with the sum is taken over all samples within the batch.

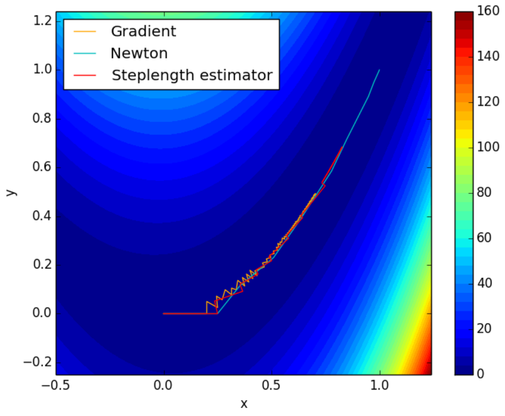

Figure 1. Comparison of iterations for minimizing the Rosenbrock function $(a = 1.0, b = 50.0)$ without noise. Comparisons are made between gradient descent, 'Gradient', gradient descent with a line search estimator, 'Steplength estimator', and full Newton scheme, 'Newton'.

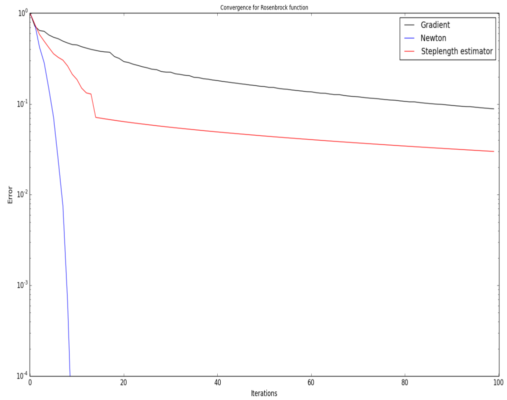

Figure 2. Convergence for minimization of the Rosenbrock function $(a = 1.0, b = 50.0)$ using, gradient descent, 'Gradient', gradient descent with a line search estimator, 'Steplength estimator', and full Newton scheme, 'Newton'.

3. Method

Succesfully training a neural network involves a number of components, an appropriate labelled dataset, the correct loss function and neural network architecture, a working solver for training the architecture and a well chosen evaluation metric. All these components must be working for the system to be successful and errors in one aspect may obscure errors in another. For example, a large dataset which is well represented by its test set may hide a poor line-search implementation. For these reasons it is desirable to incrementally test each component. Testing will be carried out for two different different models; i. Logistic regression and ii. softmax classification with a neural network. Doing this will validate different aspects of the system. Logistic regression is a convex function but is trained by the same methods as non-convex neural networks. Therefore, testing with logistic regression validates that numerical optimization is implemented correctly. It also tests the viability of the stochastic sampling method. Neural networks are non-convex, and for a large dataset require batch sampling to train efficiently. Therefore, they represent a true test of optimization performance.

3.1. Data

When training the logistic regression classifier I used handmade dataset consisting of 800 pairs of points in the (x, y) plane each with a label, 0 or 1. Validation of the classifier (Tables 1 and 2) was carried out using the validation set which consists of 100 pairs of labelled points in the (x, y) as illustrated in Figure 3.

When experimenting with the neural network classifier I used the Zalando Research (2017), Fashion MNIST dataset.

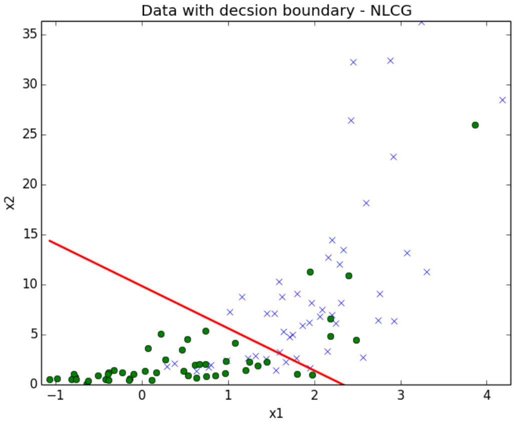

Figure 3. Validation dataset for Logistic regression with the decision boundary found by using an NLCG solver.

This dataset contains 60,000 training images and 10,000 testing images of clothing types, (labels: T-shirts, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot, with corresponding index numbered 0 - 9). Each image is 28 × 28 pixels in size, and is generally represented as a flat vector of 784 numbers. The dataset includes labels for each of the ten examples, a number indicating the actual clothing types (0 - 9) in that image. As the labels are not binary a multinomial loss function must be used.

3.2. Models

This project will test convergence of various solvers for two loss functions, logistic loss and softmax (multinomial loss). In this section I define these loss-functions in terms of their weights (models).

Logistic regression

Given data, $X \in \mathbb{R}^{(m,n)}$ consisting of m examples each with n features, and a set of m binary classes for each example, $y \in \{0,1\}^{(m)}$, then we determine a model, $\theta \in \mathbb{R}^n$, which fits X to y by minimizing the logistic-loss,

$$\mathcal{L} = -\frac{1}{m}\sum_i y^{(i)}\log(\widehat{y}^{(i)}) + (1-y^{(i)})\log(1-\widehat{y}^{(i)}). \tag{15}$$

Where,

$$\widehat{y} = h_\theta(X) = \frac{1}{1+\exp(-X\theta)}. \tag{16}$$

The gradient and Hessian of this function are given by,

$$\nabla\mathcal{L} = \frac{1}{m}X^T(h_\theta(X) - y), \tag{17}$$

$$\nabla^2\mathcal{L} = \frac{1}{m}X^T[h_\theta(X)(1-h_\theta(X))]X. \tag{18}$$

The Hessian is positive-semi definite and so $\mathcal{L}$ is convex with a well defined minimum. Hence, whilst this function is non-linear on its parameter θ an optimizer cannot become stuck in a local minima, though clearly some optimization methods will converge faster than others.

Neural Networks

For good performance on Fashion MNIST softmax regression is unlikely to be sufficient. To increase model complexity, I will use a neural network consisting of L layers with the weights of the layers being defined by a list of matrices, W $(= [W_1, W_2, \ldots, W_L])$, and biases, b $(= [b_1, b_2, \ldots, b_L])$. The forward propagation equation for the l-th layer takes the form,

$$Z_j^{[l+1](i)} = W_{j,k}^{[l+1]}X_k^{[l](i)} + b_j^{[l+1]}, \tag{19}$$

$$X_j^{[l+1](i)} = \sigma^{[l+1]}(Z_j^{[l+1](i)}), \tag{20}$$

where $\sigma^{[l]}$ denotes the choice of activation function for the l-th layer. As mentioned, in Fashion MNIST each label is an integer representing a clothing type 0 - 9 and as such y is a multinomial class (with $K = 10$) rather than binary. Therefore, at the final layer, L I will use the cross-entropy loss function,

$$\mathcal{L} = -\frac{1}{m}\sum_i\sum_{k=1}^{K} Y_k^{(i)}\log(X_k^{[L](i)}), \tag{21}$$

with

$$X^{[L]} = \sigma^{[L]}(Z^{[L]}), \tag{22}$$

where σ is the softmax function,

$$\sigma(Z_i) = \frac{\exp(Z_i)}{\sum_i \exp(Z_i)}, \tag{23}$$

and Y is the one-hot matrix representation of the multinomial classes. The final loss function is calculated using the 'forward propagation' algorithm (Algorithm 1).

Algorithm 1 Forward propagation

```
$X^{(0)} \leftarrow X$
for $l \leftarrow 1, L$ do
    $Z^{(l)} \leftarrow W^{(l)}X^{(l-1)} + b^{(l)}$
    $X^{(l)} \leftarrow \sigma(Z^{(l)})$
end for
$\mathcal{L} \leftarrow -\frac{1}{n}\mathrm{sum}\left[\log(X^{(L)}) + (1-y)Z^{(L)}\right]$
```

The gradient of the loss function with respect to the set of weights W, $\nabla\mathcal{L}$ $(= [\nabla\mathcal{L}_W^1, \nabla\mathcal{L}_W^2, \ldots, \nabla\mathcal{L}_W^L])$ are calculated by back-propagation. Starting from the final layer,

$$\left[\nabla\mathcal{L}_Z^{[L]}\right]_k^{(i)} = \frac{1}{n}\left[X_k^{[L](i)} - y_k^{(i)}\right]. \tag{24}$$

The update equation are,

$$[\nabla\mathcal{L}_W^{[l]}]_{j,k} = X_k^{[l-1](i)} \left[\nabla\mathcal{L}_Z^{[l]}\right]_j^{(i)}, \tag{25}$$

$$[\nabla\mathcal{L}_b^{[l]}]_j = \sum_i \left[\nabla\mathcal{L}_Z^{[l]}\right]_j^{(i)}. \tag{26}$$

and the backpropagation equation are,

$$\left[\nabla\mathcal{L}_X^{[l-1]}\right]_j^{(i)} = \left(W^T\right)_{j,k}^{[l]} \left[\nabla\mathcal{L}_Z^{[l]}\right]_k^{(i)}, \tag{27}$$

$$\left[\nabla\mathcal{L}_Z^{[l-1]}\right]_j^{(i)} = \sigma'(Z_j^{[l-1](i)}) \left[\nabla\mathcal{L}_X^{[l-1]}\right]_j^{(i)}. \tag{28}$$

Derivations of these equations are given in appendix A. These equation lead to the well-known 'backward propagation' algorithm (Algorithm 2). A Hessian can be calculated for each layer by taking second derivatives of equations 24-28. The calculations are cumbersome and so the derivation of these equations, together with an algorithm for computing them, is presented in appendix B.

Algorithm 2 Backward propagation

```
$\nabla\mathcal{L}_Z^{[L]} \leftarrow \frac{1}{n}\left[X^{[L]} - y\right]$
for $l \leftarrow L, 1$ do
    $\nabla\mathcal{L}_W^{[l]} \leftarrow \nabla\mathcal{L}_Z^{[l]} X^{T[l-1]}$
    $\nabla\mathcal{L}_b^{[l]} \leftarrow \mathrm{sum}(\nabla\mathcal{L}_Z^{[l]})$
    $\nabla\mathcal{L}_X^{[l-1]} \leftarrow W^{(l)T}\nabla\mathcal{L}_Z^{[l]}$
    $\nabla\mathcal{L}_Z^{[l-1]} \leftarrow \sigma'(Z^{[l-1]})\nabla\mathcal{L}_X^{[l-1]}$
end for
```

3.3. Stochastic sampling strategy

In section 4 we shall see that the stochastic sampling method has a significant influence on the success of higher-order methods. Given m samples we define a batch $\mathcal{B}$ of size, m_b, to be a randomly chosen subset of $\{1, 2, \ldots, m\}$ chosen without replacement. With this, $X^{\mathcal{B}}$, $y^{\mathcal{B}}$ denote a subsampling of the full data, X, and labels, y with elements defined by the indices of the entries in $\mathcal{B}$. The sampling is repeated a number, R, times to produce batches $\mathcal{B}_1, \ldots, \mathcal{B}_R$. From this we define a stochastic objective function suitable for experimentation with a non-linear solver,

$$\mathcal{L}_{sub}(X, y) = \frac{1}{R}\sum_{r=1}^{R} \mathcal{L}(X^{\mathcal{B}_r}, y^{\mathcal{B}_r}). \tag{29}$$

It would seem more correct to resample the batch $\mathcal{B}$ at each iteration. However, as mentioned by Martens (2010), doing this breaks the assumptions of L-BFGS or NLCG and causes early termination of the solver. Therefore, each batch is fixed throughout training. It is also unnecessary to have repeated coverings of the input data, so in practice I use, $R = \lceil m/m_b \rceil$.

3.4. Evaluation metrics

I will compare the performance of different solvers based on two metrics: the convergence of the solver during training and the bias-variance diagnostic. The former is the most important for distinguishing between solvers as it tests the rate of convergence. However, the latter is also important as a poorly chosen stochastic algorithm can converge well on a sample of the training set but have poor generalization error on the test set. It is also good practice to perform a gradient test on any loss function used. An inaccurate gradient may cause poor solver convergence, and stochastic sampling is likely to exacerbate any unresolved convergence issues. Furthermore, non-linear, non-convex objective functions are known to have saddle-points and local minima which can prevent convergence, or lead to convergence to the wrong minimum. Therefore, any bugs in the implementation risk being confused with a diagnosis of high-bias for a Neural network model.

Optimization convergence

For each solver, the training error will be plotted against the number of iterations during training. This is the key metric for discriminating between solver choices. Better solvers will converge faster. Convergence is defined as the number of iterations, k_ϵ, such that,

$$\|f(x_k) - f(x_{k-1})\| \leq \epsilon. \tag{30}$$

Bias-Variance Diagnostic

Dev (or test) accuracy can be defined by a sum over all samples within the dev (or test) set,

$$\text{accuracy}(x, y) = \frac{1}{m}\sum_{i=1}^{m}[y^{(i)} = \text{predict}(x^{(i)})]. \tag{31}$$

I will plot training error together with test accuracy for each choice of solver over a range of crucial hyper-parameters. Examples of these hyper-parameters are batch-size, m_b, regularization parameter, λ, and the number of layers in the neural network, L. For a purely convex function this test will not be informative since all solvers should converge to the same set of weights albeit at different rates. Thus, if all solvers achieve the same training error then the test error should also be the same for the same dataset. However, this is not necessarily the case when using stochastic sampling as a poorly chosen stochastic algorithm can converge well on a sample of the training set but have poor generalization error on the test set, see Table 1 for an example of this.

For a non-convex method the situation is more complicated. Here different solvers may converge to different minima and this impacts the bias-variance tradeoff. For example, if solver A becomes trapped in a local minima whereas B converges then the training error will be less for A. However, the solution for B may actually overfit the data whereas A does not. Here, A will be exhibiting high-bias, lower-variance, whereas B will show lower-bias, high-variance. In other words, B is capable of simulating a larger range of models than A, therefore is increasing variance. Furthermore, the choice of sampling method will interact with the shape of the loss-function surface obscuring or even reinforcing the problem of saddle-points and local minima. For this reason, it is important to check bias-variance curves when comparing solvers on deep networks.

GRADIENT TEST

Any objective function should be tested to ensure that for randomly chosen m and Δm and a suitably small ϵ,

$$\frac{1}{2\epsilon}(\mathcal{L}(m+\epsilon\Delta m)-\mathcal{L}(m-\epsilon\Delta m)) = \Delta m^T \nabla\mathcal{L} + O(\epsilon^2). \quad (32)$$

4. Results

4.1. Logistic regression

In Figure 4, I compare the convergence rate between i. NLCG, ii. Newtons' method with line-search, iii. Newtons' method without line-search adaption, iv. L-BFGS during training on batches of the dataset using stochastic sampling. A batch size of 10 samples was chosen and the batches were reshuffled at each iteration. The sampling was repeated 'Nbatches' times when calculating the objective function following the method of equation 29. Note that due to reshuffling of the batches the NLCG and BFGS solvers terminated quickly. The corresponding test-error for this sampling method is shown in Table 1

Solver	Train Loss	Test error
Newton (LS)	0.4019	0.83
Newton (noLS)	0.5613	0.81
L-BFGS	0.6030	**0.51**
NLCG	0.5989	**0.51**

Table 1. Number of iterations and test error for stochastic logistic regression using a batch size of 10, with sampling repeated 50 times and using reshuffling at each iteration. Note that L-BFGS and NLCG have very poor performance on the test-set.

In contrast to the reshuffling method, Figure 5 shows the convergence rate for i. NLCG, ii. Newtons' method with line-search, iii. Newtons' method without line-search adaption, iv. L-BFGS during training. A batch size of 10 samples was chosen and the batches were fixed throughout iteration. The sampling was not repeated, however 80 batches were used to cover the full training set. The corresponding test-error for this sampling method is shown in Table 2

Solver	Niters	Train Loss	Test error
Newton (LS)	8	0.4058	0.83
Newton (noLS)	**50**	**0.5617**	**0.81**
BFGS	20	0.4058	0.83
NLCG	31	0.4058	0.83

Table 2. Number of iterations ('Niters') and test error for logistic regression with a batch size of 10 using various solvers. Newton's method without line search shows higher bias due to early stopping.

4.2. Fashion MNIST

In Figure 6 and 7 I compare the convergence rate between i. Gradient descent, ii. NLCG and iii. L-BFGS-B during training on batches of Fashion MNIST. The convergence on the dev set is shown and the train-dev accuracy is also plotted. In Figure 6, one hidden layer was used, whereas in Figure 7 two hidden layers were used. For these tests, each batch of training data had a size of 1000 x 784, and when using one hidden layer the weights and biases had a sizes of 784 x 300 and 300 respectively. The output layer had a size of 301 x 10. With two hidden layers the weights and biases had sizes of 784 x 300 and 300 (first layer), 300 x100 and 100 (second layer), and 101 x 10 for the output layer. A batch size of 1000 samples was chosen and the batches were fixed throughout each iteration. Whilst the train and dev curves are surprisingly close they are not exactly the same, as is shown in Table 3. Comparisons of the runtime per iteration for each of these solvers is shown in Table 4.

Solver	Train Acc	Dev Acc
Gradient (1L)	0.7232	0.7195
NLCG (1L)	0.8363	0.8336
L-BFGS-B (1L)	0.8307	0.8293
Gradient (2L)	0.7029	0.7038
NLCG (2L)	0.7627	0.7650
L-BFGS-B (2L)	0.7782	0.7845

Table 3. Train and Dev accuracy using 100 iterations on Fashion MNIST. L stands for the number of hidden layers.

Solver	Runtime (s)	Runtime (s) per iteration
Gradient (1L)	411.9	8.2
L-BFGS-B (1L)	569.3	11.3
NLCG (1L)	751.1	15.0
Gradient (2L)	509.7	10.2
L-BFGS-B (2L)	1010.1	20.2
NLCG (2L)	1215.2	24.3

Table 4. Runtime per iteration for 50 iterations with one (1L) or two (2L) hidden layers and regularization, $\lambda = 0.01$.

4.3. Convexity of Neural Networks

In section 4.2 I avoided testing for a full Newton solver even though the networks were reasonably shallow. Irrespective of any efficiency concerns, I deemed the equations in appendix B too cumbersome to be implemented quickly and accurately. However, they can still yield important insights into the nature of the loss function surface. For example, we can obtain the following result,

Theorem 4.1. *If sigmoid activations are used at each hidden layer, and*

$$\text{sign}\left(Z_j^{[l](i)}\right) = \text{sign}\left(\nabla\mathcal{L}_{Z,j}^{[l](i)}\right) \ \forall i, j, \tag{33}$$

then the loss function is convex.

Proof. A result of convex analysis states that an objective function, $f : \mathbb{R}^n \to \mathbb{R}$, is convex if and only if its Hessian is positive semi-definite (PSD), for details see appendix C, or page 71 of Boyd and Vandenberghe (2004),

$$f \text{ is convex } \iff \mathbf{x}^T \nabla^2 f \mathbf{x} \geq 0, \quad \forall \mathbf{x} \in \mathbb{R}^n. \tag{34}$$

We note from equation 69 that $H^{[l]}$ is PSD if and only if each $H_Z^{[l]}$ is. That $H_Z^{[l]}$ is PSD follows from induction on the number of layers counting backwards from the output layer L. $H_Z^{[L](i)}$ is PSD since,

$$\begin{aligned}\mathbf{y}^T H_Z^{[L](i)} \mathbf{y} &= \frac{1}{m} \sum_p X_p^{[L](i)} y_p^2 - (\sum_q X_p^{[L](i)} y_p)^2, \\ &= \frac{1}{m} \sum_p X_p^{[L](i)} (y_p - \sum_q X_q^{[L](i)} y_q)^2 \quad (35)\end{aligned}$$

The last expression is ≥ 0 since $X_p^{[L](i)} \geq 0$ (see equations 22 and 23).

For the inductive step, assume the result holds at layer l. Then, from equations 71 and 72, we find for layer $l-1$,

$$\begin{aligned}&\mathbf{y}^T H_Z^{[l-1](i)} \mathbf{y} = \\ &\mathbf{y}^T \sigma'(Z^{[l-1](i)}) \circ W^{T[l]} (H_Z^{[l](i)}) W^{[l]} \circ \sigma'(Z_{k'}^{[l-1](i)}) \mathbf{y} \\ &+ \mathbf{y}^T \sigma''(Z^{[l-1](i)}) \circ \left(\nabla\mathcal{L}_X^{[l-1](i)}\right) \mathbf{y}. \quad (36)\end{aligned}$$

The first term in the sum on the right hand side is of the form,

$$\mathbf{y}'^T (H_Z^{[l](i)}) \mathbf{y}', \tag{37}$$

and by the inductive hypothesis,

$$\mathbf{y}'^T (H_Z^{[l](i)}) \mathbf{y}' \geq 0. \tag{38}$$

The second term in the sum will also be positive whenever,

$$\sigma''(Z_j^{[l-1](i)}) \left(\nabla\mathcal{L}_X^{[l-1]}\right)_j^{(i)} \geq 0. \tag{39}$$

This occurs whenever,

$$\text{sign}\left(Z_j^{[l-1](i)}\right) = \text{sign}\left(\nabla\mathcal{L}_{X,j}^{[l-1](i)}\right), \tag{40}$$

and from equation 52,

$$\text{sign}\left(\nabla\mathcal{L}_{X,j}^{[l-1](i)}\right) = \text{sign}\left(\nabla\mathcal{L}_{Z,j}^{[l-1](i)}\right). \tag{41}$$

□

Theorem 4.1 can provide a means of enforcing convexity during back-propagation.

Corollary 4.2. *If equation 33 fails to hold at layer l for some i, j then we can set,*

$$\nabla\mathcal{L}_{Z,j}^{[l](i)} = 0. \tag{42}$$

This says that we can find a convex approximate Hessian suitable as a preconditioner for training the Neural network by filtering the gradient, $\nabla\mathcal{L}_Z$, to zero whenever,

$$\text{sign}\left(Z^{[l]}\right) \neq \text{sign}\left(\nabla\mathcal{L}_Z^{[l]}\right). \tag{43}$$

5. Discussion

Figure 2 shows that second-order solvers have the ability to greatly outperform gradient descent. Figure 5 shows that provided fixed batches are used, second-order schemes with line-search outperform Newton methods without line searching. Indeed, the drastic improvement in convergence between using a line-search and using fixed step-size is striking. The explanation is that even with an accurate downhill step, if no adjustment of the step-size is performed by using a line search together with Wolfe conditions (see Nocedal and Wright, 2006), then the step-length can be inappropriate leading to slow convergence.

It is interesting that rapid convergence was restored once resampling of the batches was abandoned and fixed batches were used. To understand this, recall that logistic regression does not contain any hidden layers. Therefore, batch sampling (with fixed batches) followed by regathering the loss function over all batches into the final objective function (equation 29), is equivalent to using all batches in a single objective function. The presence of a hidden layer adds non-linearity and breaks this equivalence.

For the case of training neural networks on Fashion MNIST the situation was less clear cut. It is noticeable that in early iterations the convergence was unstable (most likely the influence of batch sampling) and convergence took a while to 'settle down'. However, there was a point when convergence for both L-BFGS and NLCG beat gradient descent and eventually settled down to convergence within

100 iterations, whilst gradient descent still had some way to go. Typically stochastic gradient descent schemes take upwards of 10^4 iterations. Eventually, an accuracy of 0.83 was achieved for both second-order solvers on both the train and dev sets. It is also worrying that the statistics for the accuracy of the model on both the train and dev sets were so close. It is possible that this is a feature of the stochastic sampling method, but this matter needs further investigation as this could also be an error. The runtime of NLCG per iteration was almost twice that of gradient descent with L-BFGS-B coming in someway between. This higher cost per iteration might be considered less significant compared to the benefits gained by faster convergence, ie requiring far fewer iterations.

Acnowledgements

The author would like to thank Andrey Koch for his encouragement and helpful suggestions throughout this project.

Code availability

All codes used during the compilation of this paper are available through the GitHub project, https://github.com/bveitch/StochOptDL

References

H. Asi and J.C. Duchi. The importance of better models in stochastic optimization. *Proceedings of the National Academy of Sciences of the United States of America*, 116 (46):22924–22930, 2019.

S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, 2004.

J. Collins, J. Sohl-Dickstein, and D. Sussillo. Capacity and trainability in recurrent neural networks. arXiv:1611.09913[stat.ML], 2016.

J.C. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, E. Mortensen, K. Saenko, Eds*, page 770– 778, IEEE, Piscataway, NJ, 2016, 2016.

G.E. Hinton. Lecture 6.1 — Overview of mini-batch gradient descent — [Deep Learning — Geoffrey Hinton — UofT], 2017. YouTube: https://www.youtube.com/watch?v=EANflep-cog.

J. Hong and P. Wilford. A Stochastic Conjugate Gradient Method for Approximation of Functions. https://arxiv.org/pdf/1302.1945.pdf, 2013.

G. Huang, Y. Sun, L. Zhuang, D. Sedra, and K. Weinberger. Deep networks with stochastic depth. https://arxiv.org/pdf/1603.09382v2.pdf, 2016.

D.P. Kingma and J. Lei Ba. Adam: a method for stochastic optimization. In *International Conference on Learning Representations*, page 1–13, 2015.

Q.V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A.Y. Ng. On Optimization Methods for Deep Learning. In *Proceedings of the 28th International Conference on Machine Learning*, Bellevue, WA, USA, 2011.

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

J. Martens. Deep learning via Hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning*, Haifa, Israel, 2010.

J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer-Verlag, Berlin, New York, 2006.

N. Qian. On the momentum term in gradient descent learning algorithms. *Neural networks : the official journal of the International Neural Network Society*, 12(1):145–151, 1999.

S. Ruder. An overview of gradient descent optimization algorithms. https://arxiv.org/pdf/1609.04747v2.pdf, 2017.

SciPy Team. *scipy.optimize.minimize, SciPy 1.7.1*. docs.scipy.org, 2020. https://docs.scipy.org/doc/scipy/reference/optimize.html.

J. R. Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1994.

N. Srebro and A. Tewari. Tutorial: Stochastic Optimisation for Machine Learning. In *Proceedings of the 27th International Conference on Machine Learning*, Haifa, Israel, 2010.

P. Xu, F. Roosta-Korasani, and M.W. Mahoney. Second-Order Optimization for Non-Convex Machine Learning: An Empirical Study. https://arxiv.org/pdf/1708.07827v2.pdf, 2018.

Zalando Research. Fashion MNIST dataset, 2017. https://github.com/zalandoresearch/fashion-mnist.

R. Zhao, W.B. Haskell, and V.Y.F. Tan. Stochastic L-BFGS: Improved Convergence Rates and Practical Acceleration Strategies. *IEEE Transactions on Signal Processing*, 66: 1155 – 1169, 2018.

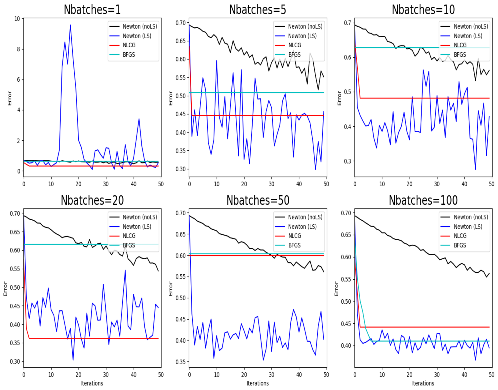

Figure 4. Convergence of logistic regression during training with mini-batch sampling for NLCG, Newtons' method with line-search, Newton (LS), Newtons' method without line-search adaption, Newton (noLS) and L-BFGS. A batch size of 10 samples was chosen and 'Nbatches' refers to the number of times the sampling is repeated (with reshuffling) when calculating the objective function defined in equation 29.

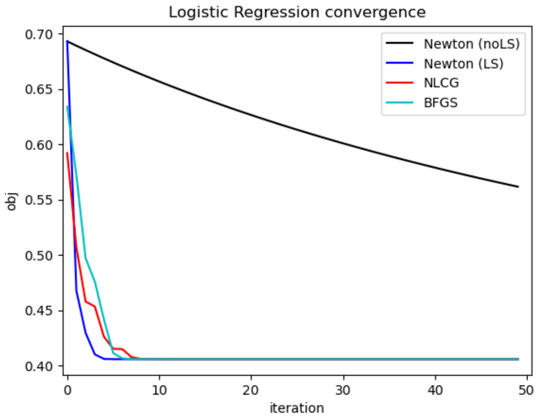

Figure 5. Convergence of logistic regression during training with mini-batch sampling of 10 samples for NLCG, Newtons' method with line-search, Newton (LS), Newtons' method without line-search adaption, Newton (noLS) and L-BFGS. Here the batches were fixed throughout training and $800/10 = 80$ batches were used.

A. Derivation of backward propagation equations

Starting from the definition of the cross-entropy loss function 21,

$$\left(\frac{\partial \mathcal{L}}{\partial X}\right)_k^{[L](i)} = -\frac{1}{m}\frac{y_k^{(i)}}{X_k^{[L](i)}},$$

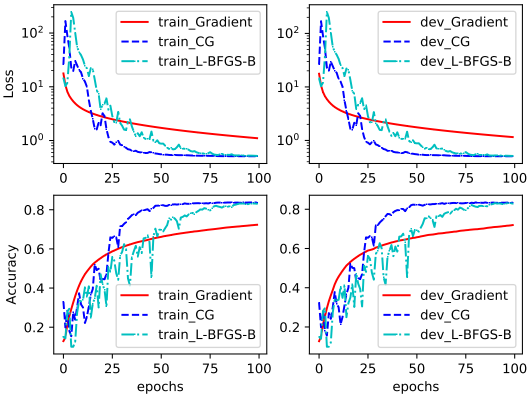

Figure 6. Convergence of neural network during training on Fashion MNIST. Training was performed using one hidden layer, for 100 epochs with a batch size of 1000 samples and a regularization parameter, $\lambda = 0.01$.

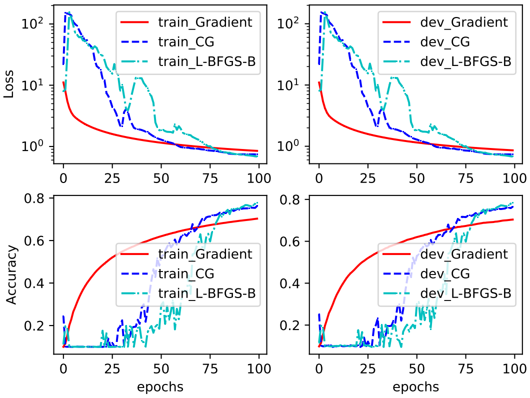

Figure 7. Convergence of neural network during training on Fashion MNIST. Training was performed using two hidden layers, for 100 epochs with a batch size of 1000 samples and a regularization parameter, $\lambda = 0.01$.

and so, by the chain-rule the derivatives with respect to Z are,

$$\left(\frac{\partial \mathcal{L}}{\partial Z}\right)_k^{[L](i)} = \left(\frac{\partial \mathcal{L}}{\partial X}\right)_r^{[L](i)} \left(\frac{\partial X_r^{[L](i)}}{\partial Z_k^{[L](i)}}\right),$$

with,

$$\left(\frac{\partial X_r^{[L](i)}}{\partial Z_k^{[L](i)}}\right) = X_r^{[L]i}(\delta_{k,r} - X_k^{[L]i}).$$

Therefore,

$$\left(\frac{\partial \mathcal{L}}{\partial Z}\right)_k^{[L](i)} = \frac{1}{m}\left[X_k^{[L](i)} - y_k^{(i)}\right].$$

Using the chain-rule for the weights derivatives,

$$\left(\frac{\partial \mathcal{L}}{\partial W_{j,k}}\right)^{[l]} = \sum_{i,p}\left(\frac{\partial \mathcal{L}}{\partial Z_p}\right)^{[l](i)}\left(\frac{\partial Z_p^{[l](i)}}{\partial W_{j,k}^{[l]}}\right), \tag{44}$$

$$\left(\frac{\partial \mathcal{L}}{\partial b_j}\right)^{[l]} = \sum_{i,p}\left(\frac{\partial \mathcal{L}}{\partial Z_p}\right)^{[l](i)}\left(\frac{\partial Z_p^{[l](i)}}{\partial b_j^{[l]}}\right). \tag{45}$$

By equation 19,

$$\frac{\partial Z_p^{[l](i)}}{\partial W_{j,k}^{[l]}} = \delta_{p,j} X_k^{[l-1](i)}, \tag{46}$$

$$\frac{\partial Z_p^{[l](i)}}{\partial b_j^{[l]}} = \delta_{p,j}, \tag{47}$$

$$\left(\frac{\partial Z_p^{[l]}}{\partial X_k^{[l-1]}}\right)^{(i)} = W_{p,k}^{[l]}. \tag{48}$$

Substituting 46 into 44, and 47 into 45 leads to,

$$\begin{aligned}\left(\frac{\partial \mathcal{L}}{\partial W_{j,k}}\right)^{[l]} &= \sum_i \left(\frac{\partial \mathcal{L}}{\partial Z_j}\right)^{[l](i)} X_k^{[l-1](i)} \\ &= X_k^{[l-1],T}\left(\frac{\partial \mathcal{L}}{\partial Z_j}\right)^{[l]},\end{aligned} \tag{49}$$

$$\left(\frac{\partial \mathcal{L}}{\partial b_j}\right)^{[l]} = \sum_i \left(\frac{\partial \mathcal{L}}{\partial Z_j}\right)^{[l](i)}. \tag{50}$$

Whilst from equation 48,

$$\begin{aligned}\left(\frac{\partial \mathcal{L}}{\partial X_k}\right)^{[l-1](i)} &= \sum_p \left(\frac{\partial \mathcal{L}}{\partial Z_p}\right)^{[l](i)}\left(\frac{\partial Z_p^{[l](i)}}{\partial X_k^{[l-1](i)}}\right), \\ &= \sum_p W_{p,k}^{[l]}\left(\frac{\partial \mathcal{L}}{\partial Z_p}\right)^{[l](i)}, \\ &= (W^T)_{k,p}^{[l]}\left(\frac{\partial \mathcal{L}}{\partial Z_p}\right)^{[l](i)}.\end{aligned} \tag{51}$$

Finally, from equation 20,

$$\begin{aligned}\left(\frac{\partial \mathcal{L}}{\partial Z_k}\right)^{[l-1](i)} &= \sum_p \left(\frac{\partial \mathcal{L}}{\partial X_p}\right)^{[l-1](i)}\left(\frac{\partial X_p^{[l-1](i)}}{\partial Z_k^{[l-1](i)}}\right), \\ &= \sigma'(Z_k^{[l-1](i)})\left(\frac{\partial \mathcal{L}}{\partial X_k}\right)^{[l-1](i)}\end{aligned} \tag{52}$$

Equations 51 and 52 are the back-propagation equations and used in equations 27 and 28 within the main text. Equations 49 and 50 define the updates for the weights and biases, as used in equations 25 and 26.These equations lead to the back-propagation algorithm shown in algorithm 2

B. Second-order derivatives

We continue the backward propagation equations to calculate the second derivatives. If we restrict attention to finding derivatives only between the hidden units in each layer rather than between layers, then the Hessian matrix for layer l takes the form,

$$H^{[l]} = \begin{bmatrix} H_W^{[l]} & H_{Wb}^{[l]} \\ H_{bW}^{[l]} & H_b^{[l]} \end{bmatrix}, \tag{53}$$

in which,

$$(H_W^{[l]})_{jk,j'k'} = \frac{\partial^2 \mathcal{L}}{\partial W_{j,k}^{[l]} \partial W_{j',k'}^{[l]}} \tag{54}$$

$$(H_b^{[l]})_{j,j'} = \frac{\partial^2 \mathcal{L}}{\partial b_j^{[l]} \partial b_{j'}^{[l]}} \tag{55}$$

$$(H_{Wb}^{[l]})_{jk,j'} = \frac{\partial^2 \mathcal{L}}{\partial W_{j,k}^{[l]} \partial b_{j'}^{[l]}} \tag{56}$$

$$(H_{bW}^{[l]})_{j,j'k'} = \frac{\partial^2 \mathcal{L}}{\partial b_j^{[l]} \partial W_{j',k'}^{[l]}}. \tag{57}$$

Under these definitions a Hessian for each layer can be calculated from the backpropagation equations using the second-order derivatives,

$$(H_W^{[l]})_{jk,j'k'} = (H_Z^{[l]})_{j,j'}^{(i)} X_k^{[l-1](i)} X_{k'}^{[l-1](i)}, \tag{58}$$

$$(H_b^{[l]})_{j,j'} = \sum_i (H_Z^{[l]})_{j,j'}^{(i)}, \tag{59}$$

$$(H_{Wb}^{[l]})_{jk,j'} = \sum_i X_k^{[l-1](i)} (H_Z^{[l]})_{j,j'}^{(i)}, \tag{60}$$

$$(H_{bW}^{[l]})_{j,j'k'} = \sum_i (H_Z^{[l]})_{j,j'}^{(i)} X_{k'}^{[l-1](i)}, \tag{61}$$

where,

$$(H_Z^{[l]})_{j,j'}^{(i)} = \frac{\partial^2 \mathcal{L}}{\partial Z_j^{[l](i)} \partial Z_{j'}^{[l](i)}}. \tag{62}$$

Equations 58 to 62 give rise to an expression for the Hessian for each layer in the matrix form,

$$H_W^{[l]} = (I_{n_l,n_l} \otimes X^{[l-1]}) \widehat{H}_Z^{[l]} (I_{n_l,n_l} \otimes X^{[l-1]})^T, \tag{63}$$

$$H_{Wb}^{[l]} = (I_{n_l,n_l} \otimes X^{[l-1]}) \widehat{H}_Z^{[l]} (I_{n_l,n_l} \otimes \mathbf{1}_{n_l})^T, \tag{64}$$

$$H_{bW}^{[l]} = (I_{n_l,n_l} \otimes \mathbf{1}_{n_l}) \widehat{H}_Z^{[l]} (I_{n_l,n_l} \otimes X^{[l-1]})^T, \tag{65}$$

$$H_b^{[l]} = (I_{n_l,n_l} \otimes \mathbf{1}_{n_l}) \widehat{H}_Z^{[l]} (I_{n_l,n_l} \otimes \mathbf{1}_{n_l})^T, \tag{66}$$

where,

$$\widehat{H}_Z^{[l]} = \mathrm{Diag}\,(H_Z^{[l](0)}, H_Z^{[l](1)}, \ldots, H_Z^{[l](m)}), \tag{67}$$

and $A \otimes B$ denotes the tensor product,

$$A \otimes B = \begin{bmatrix} Ab_{11} & Ab_{12} & \ldots & Ab_{1n} \\ Ab_{21} & Ab_{22} & \ldots & Ab_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ Ab_{m1} & Ab_{m2} & \ldots & Ab_{mn} \end{bmatrix}. \tag{68}$$

Consequently, the full Hessian matrix can be written in the form,

$$H^{[l]} = \widehat{X}^{[l-1]} \widehat{H}_Z^{[l]} \widehat{X}^{[l-1]T}, \tag{69}$$

with,

$$\widehat{X}^{[l-1]} = \left[\begin{array}{cc} (I_{n_l,n_l} \otimes X^{[l-1]}) & 0 \\ 0 & (I_{n_l,n_l} \otimes \mathbf{1}_{n_l}) \end{array} \right]. \tag{70}$$

It remains to determine the back-propagation equations for the inner-matrices, $H_Z^{[l](i)}$, we find that in moving from layer l to $l-1$ we have

$$(H_X^{[l-1](i)})_{k,k'} = \left(W^T\right)^{[l]}_{k,p} (H_Z^{[l](i)})_{p,q} W^{[l]}_{q,k'}. \tag{71}$$

$$(H_Z^{[l-1](i)})_{k,k'} = \sigma'(Z_k^{[l-1](i)})(H_X^{[l-1](i)})_{k,k'}\sigma'(Z_{k'}^{[l-1](i)}) + \sigma''(Z_k^{[l-1](i)})\delta_{k,k'} \left(\frac{\partial \mathcal{L}}{\partial X_k}\right)^{[l-1](i)}. \tag{72}$$

Where the propagation starts from layer L where,

$$H_{Z,p,q}^{[L](i)} = \frac{1}{m} X_p^{[L](i)} \left(\delta_{p,q} - X_q^{[L](i)}\right). \tag{73}$$

Equations 69, 71 and 72 define an algorithm for back-propagating the Hessian from layer L to 1.

C. Convexity and positive semi-definiteness

For completeness we include a proof of the theorem,

Theorem C.1. *A smooth objective function f, $f : \mathbf{R}^n \to \mathbf{R}$ is convex if and only if its Hessian, $\nabla^2 f$, is positive semi-definite.*

First, we will require a preparatory lemma,

Lemma C.2. *Let $f : \mathbf{R}^n \to \mathbf{R}$ and let $\mathcal{D}$ be a convex subset of $\mathbf{R}^n$. Then, f is convex if and only if for any $\mathbf{x}, \mathbf{y} \in \mathcal{D}$ we have,*

$$f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x}). \tag{74}$$

Proof. (only if) Let $\mathbf{z} = \lambda\mathbf{y} + (1-\lambda)\mathbf{x}$, for $\lambda \in [0,1]$, and assuming f is convex,

$$f(\mathbf{z}) \leq \lambda f(\mathbf{y}) + (1-\lambda) f(\mathbf{x}), \tag{75}$$

and

$$f(\mathbf{z}) - f(\mathbf{x}) \leq \lambda(f(\mathbf{y}) - f(\mathbf{x})). \tag{76}$$

On the other hand,

$$\begin{aligned} \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x}) = \lim_{h\to 0} \frac{f(\mathbf{x} + \lambda(\mathbf{y} - \mathbf{x})) - f(\mathbf{x})}{\lambda} &= \lim_{h\to 0} \frac{f(\mathbf{z}) - f(\mathbf{x})}{\lambda} \\ \leq f(\mathbf{y}) - f(\mathbf{x}). \end{aligned} \tag{77}$$

(if) Now we assume that,

$$f(\mathbf{y}) \geq f(x) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x}), \quad \forall \mathbf{x}, \mathbf{y} \in \mathcal{D}, \tag{78}$$

to show f is convex, let $\mathbf{z} = \lambda\mathbf{y} + (1-\lambda)\mathbf{x}$, for $\lambda \in [0,1]$. Since $\mathcal{D}$ is convex, $z \in \mathcal{D}$, and using 78 we have,

$$f(\mathbf{y}) \geq f(\mathbf{z}) + \nabla f(\mathbf{z})^T(\mathbf{y} - \mathbf{z}), \tag{79}$$

$$f(\mathbf{x}) \geq f(\mathbf{z}) + \nabla f(\mathbf{z})^T(\mathbf{x} - \mathbf{z}). \tag{80}$$

$\lambda\times$ (79) $+(1-\lambda)\times$(80) $\Longrightarrow$

$$\lambda f(\mathbf{y}) + (1-\lambda) f(\mathbf{x}) \geq f(\mathbf{z}). \tag{81}$$

□

Now we are in a position to prove Theorem C.1, *Proof.* (only if) By the multivariate Taylor's theorem, $\exists z \in [\mathbf{x}, \mathbf{y}]$,

$$f(\mathbf{y}) = f(\mathbf{x}) + \nabla f^T(\mathbf{y} - \mathbf{x}) + \frac{1}{2}(\mathbf{y} - \mathbf{x})\nabla^2 f(\mathbf{z})(\mathbf{y} - \mathbf{x}), \tag{82}$$

and by positive-definiteness of $\nabla^2 f$,

$$f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f^T(\mathbf{y} - \mathbf{x}). \tag{83}$$

So, Lemma C.2 $\implies$ f is convex.

(if) Given $\mathbf{x} \in \mathcal{D}$ and an arbitrary direction $\mathbf{v} \in \mathbf{R}^n$, we can define the 1D function,

$$g(t) = f(\mathbf{x} + t\mathbf{v}), \tag{84}$$

Clearly, g is smooth and convex for all t in some interval $[0, \epsilon]$ since f is smooth and convex in $\mathcal{D}$. Further,

$$g''(t) = \mathbf{v}^T \nabla^2 f(\mathbf{x} + t\mathbf{v})\mathbf{v}. \tag{85}$$

Since g is a 1D convex function, $g'' \geq 0$ for $t \in [0, \epsilon]$ and hence $\nabla^2 f$ is positive semi-definite.

□