Supplementary Material for:
Curvature-Aware Training for Coordinate Networks

Hemanth Saratchandran*[1] Shin-Fang Chng*[1] Sameera Ramasinghe[2]
Lachlan MacDonald[1] Simon Lucey[1]
[1] Australian Institute of Machine Learning, University of Adelaide.
[2] Amazon, Australia.

1. Second order optimisers

In this section we give details on various second order optimisers that were discussed in the paper. Namely, we will give details for Newton's method, the Newton Conjugate Gradient method (Newton CG), the BFGS algorithm, L-BFGS algorithm and K-FAC algorithm.

1.1. Newtons Method

Given an objective function f of$n$ variables, the second order Taylor series around a fixed point x_0 is given by

$$f(x) \approx f(x_0) + (x - x_0)\nabla f(x_0) + \frac{1}{2}(x - x_0)^T H(x_0)(x - x_0), \tag{1}$$

where $H(x_0)$ denotes the Hessian of f at the point x_0. Differentiating equation (1) and solving for a critical point x^* gives

$$x^* = x_0 - H(x_0)^{-1}\nabla f(x_0). \tag{2}$$

The Newton algorithm performs (2) iteratively as its update step

$$x_{t+1} = x_t - H(x_t)^{-1}\nabla f(x_t) \tag{3}$$

to move the function towards its global minimum. In the case that f is convex, convergence to a global minimum is guaranteed [5], however if f is non-convex then convergence is not guaranteed. Futhermore, the algorithm requires the storing of the full Hessian of f at each iteration making it have computational complexity $\mathcal{O}(n^3)$ [5]. The general algorithm is shown in algorithm 1.

Algorithm 1 Newton's Method

Require: initial point x_0
while stopping criterion not met **do**
  $g \leftarrow \nabla f$
  $H \leftarrow \nabla^2 f$
  $x \leftarrow x - H^{-1}g$
end while

[1] *Equal contribution. Correspondence to: Hemanth Saratchandran <hemanth.saratchandran@adelaide.edu.au>, Shin-Fang Chng <shinfang.chng@adelaide.edu.au>. Source code available at https://github.com/sfchng/curvature-aware-INRs.git

1.2. Newton Conjugate Gradient (Newton CG)

The main disadvantages of Newton's method is the high computational complexity involved in computing and storing the inverse Hessian of the objective function. The key idea of the Newton CG method is that one can avoid direct computation of the Hessian by following so called conjugate directions. In this way, it can be seen as an optimiser that is in-between Newton's method and first order gradient decent.
In various gradient decent algorithms, a line search is carried out to find the step size ϵ in the direction of the negative gradient. This leads to an adaptive step size, that is optimal at each iteration, and thus can lead to faster convergence. However, there is a disadvantage of such an approach to the step size of gradient decent. If we let f denote the objective function and define $g_t = \nabla f(x_t)$ to be the gradient at iteration t. Then a line search will seek to minimise f in the direction of $-g_t$. Once the line search terminates the following equations holds

$$-g_t \cdot \nabla f = 0. \tag{4}$$

The search direction at iteration $t+1$ is then the gradient at the point where the line search at iteration t terminated. This implies $g_t \cdot g_{t+1} = 0$. In other words, future search directions are orthogonal. This has the effect that the minimum in previous gradient directions is not preserved, undoing the previous work the algorithm did to find the minimum direction. Thus the algorithm has to re-minimise the objective function f in previous directions. In other words, by following the gradient at the end of each line search, we are undoing progress we made in the direction of the previous line search.
The main point of Newton CG is to solve the above orthogonality problem. Let us denote the search direction at iteration t by d_t. The point of Newton CG is to choose a search direction at iteration t that satisfies

$$d_t = -\nabla f + \beta_t d_{t-1} \tag{5}$$

where β_t is a parameter that determines how much of the previous search direction we should add back to the current direction. Two direction d_t and d_{t-1} are said to be conjugate if the following holds: $d_t^T H d_{t-1} = 0$, where H denotes the Hessian matrix. By picking the parameter β_t appropriately, the conjugate condition can be enforced. The two most common ways of choosing β_t are:

1. Fletcher-Reeves:
$$\beta_t = \frac{\nabla_\theta f(\theta_t)^T \nabla_\theta f(\theta_t)}{\nabla_\theta f(\theta_{t-1})^T \nabla_\theta f(\theta_{t-1})} \tag{6}$$

2. Polak-Ribière:
$$\beta_t = \frac{(\nabla f(\theta_t) - \nabla f(\theta_{t-1}))^T \nabla f(\theta_t)}{\nabla f(\theta_{t-1})^T \nabla f(\theta_{t-1})}. \tag{7}$$

In the experiments in the main paper we used Newton CG with the Polak-Ribière method.

Algorithm 2 Newton's Conjugate Gradient

Require: initial point θ_0
  $\rho_0 \leftarrow 0$
  $g_0 \leftarrow 0$
  $t \leftarrow 1$
  while stopping criterion not met **do**
  Choose mini-batch of examples $\{x^{(i)}, y^{(i)}\}_{i=1}^m$ randomly
    $g_t \leftarrow \nabla f$
  compute β_t via Polak-Ribière method
    $\rho_t \leftarrow -g_t + \beta_t \rho_{t-1}$
  Line search: $\epsilon^* \leftarrow arg\min_\epsilon \frac{1}{m}\sum_{i=1}^m f((x^{(i)}; \theta_t + \epsilon\rho_t), y^{(i)})$
    $\theta_{t+1} \leftarrow \theta_t + \epsilon^* \rho_t$
  end while

1.3. Broyden-Fletcher-Goldfarb-Shanno Algorithm (BFGS)

The BFGS algorithm seeks to achieve the advantages of Newton's method without the computational complexity of computing the exact Hessian. Such methods are known as Quasi-Newton methods. The main problem with Newton's method is the calculation of the Hessian, which has complexity $\mathcal{O}(n^2)$ for an objective function of n parameters, and the inversion of the Hessian, which has complexity $\mathcal{O}(n^3)$. The main idea of a Quasi-Newton method is to approximate the Hessian without explicitly calculating all the second derivatives and having to do an inversion.
The approach of the BFGS algorithm is to iteratively compute a positive definite matrix approximation M_t to the inverse Hessian. The motivation for the approach comes from considering a quadratic model for the objective function f

$$Q_t(p) = f_t + \nabla f_t^T p + \frac{1}{2} p^T M_t p \tag{8}$$

where M_t is a positive definite matrix that will be updated at each iteration t. A simple computation shows that $Q_t(0) = f_t(0)$ and $\nabla Q_t(0) = \nabla f_t(0)$. The minimiser of the above problem is given by $p_t = -M_t^{-1} \nabla f_t$, and is used as the search direction for the next iterate, which is given by

$$x_{t+1} = x_t + \alpha_t p_t \tag{9}$$

where the step length α_t is chosen so that it satisfies the Wolf conditions

$$f(x_t + \alpha_t p_t) \leq f(x_t) + c_1 \alpha_t \nabla f_t^T p_t \tag{10}$$

$$\nabla f(x_t + \alpha_t p_t)^T p_t \geq c_2 \nabla f_t^T p_t. \tag{11}$$

The BFGS method then imposes the following condition on M_t

$$M_{t+1} s_t = y_t \tag{12}$$

where

$$s_t = x_{t+1} - x_t \text{and } y_t = \nabla f_{t+1} - \nabla f_t. \tag{13}$$

Equation (12) is known as the Secant equation. Given the displacements s_t and the change of gradients y_t, the secant equation requires that the symmetric positive definite matrix M_{t+1} maps s_t into y_t. This will happen only if s_t and y_t satisfy the curvature condition:

$$s_t^T y_T > 0. \tag{14}$$

When the objective function f is strongly convex the curvature condition is always satisfied for any two points x_{t+1} and x_t. However, when f is not strongly convex the curvature condition will not always hold and needs to be explicitly enforced. This is done by enforcing conditions on the line search conditions that choose the step length α.
When the curvature condition (14) is satisfied. An approximation M_t to the inverse Hessian of f at x_t is given by the closed form formula

$$M_{t+1} = (I - \rho_t s_t y_t^T) M_t (I - \rho_t y_t s_t^T) + \rho_t s_t s_t^T. \tag{15}$$

All that is required is for the initial approximation M_0 to be chosen. In general, there is no exact method for choosing the initialisation M_0. In general, one chooses M_0 based on specific information about the problem or one simply sets it to be a multiple of the identity.

Algorithm 3 BFGS algorithm

Require: initial point x_0, convergence tolerance ϵ and initial inverse Hessian approximation M_0
  $t \leftarrow 0$
  while $||\nabla f_t|| > \epsilon$ **do**
    compute search direction: $p_t = -M_t \nabla f_t$
    set $x_{t+1} = x_t + \alpha_t \rho_t$ where α_t is computed from a line search procedure to satisfy the Wolf conditions (10).
    Define $s_t = x_{t+1} - x_t$ and $y_t = \nabla f_{t+1} - \nabla f_t$.
    Compute M_{t+1} by means of (15);
    $t \leftarrow t + 1$
  end while

1.4. Limited memory BFGS (L-BFGS)

One of the disadvantages of the BFGS algorithm is that in order to compute the Hessian inverse approximation, M_{t+1}, at iteration $t+1$ it is required to use the previos Hessian inverse approximation M_t at iteration t. This means at each iteration, the Hessian inverse approximation from the previous iteration must be stored in memory. Since the inverse Hessian M_t will generally be a dense matrix, the cost of storing it and manipulating it will be prohibitive.
The main point of L-BFGS is to circumvent this memory issue of BFGS by storing a modified version of M_t at each iteration, that requires much less memory. As in the BFGS algorithm, we define

$$s_t = x_{t+1} - x_t \text{and } y_t = \nabla f(x_{t+1}) - \nabla f(x_t). \tag{16}$$

Assume the last m updates of the pairs $\{s_t, y_t\}_{t=1}^m$ are stored. The L-BFGS algorithm starts with an initial approximation at iteration t given by $M_t^0 = \gamma_t I$, where I is the identity matrix and $\gamma_t = \frac{s_{t-1}^T y_{t-1}}{y_{t-1}^T y_{t-1}}$. Define a sequence of m scalars ρ_i by

$$\rho_i = \frac{1}{y_i^T s_i} \tag{17}$$

for $t - m \leq i \leq t$. The L-BFGS algorithm is given in Algorithm 4.

Algorithm 4 L-BFGS algorithm

Require: initial point x_0, convergence tolerance ϵ
  $t \leftarrow 1$
  while stopping criterion not met **do**
    $q \leftarrow \nabla f(x_t)$
    for $i = t-1, \cdots, t-m$ **do**
      $\alpha_i \leftarrow \rho_i s_i^T q$
      $q \leftarrow q - \alpha_i y_i$
    end for
    Compute γ_t
    $M_t^0 \leftarrow \gamma_t I$
    $z \leftarrow M_t^0 q$
    for $i = t-m, \cdots, t-1$ **do**
      $\beta_i \leftarrow \rho_i y_i^T z$
      $z \leftarrow (\alpha_i - \beta_i) s_i$
    end for
    $x_{t+1} \leftarrow x_t \epsilon z$
    $t \leftarrow t+1$
  end while

1.5. Kronecker-Factored Approximate Curvature (K-FAC)

K-FAC is a second order algorithm for neural networks that seeks to approximate the Hessian matrix in Newton's method via the Fisher information matrix F [4]. Since the Fisher information matrix is in general a dense matrix, F is further approximated by a block diagonal matrix, where each block is obtained from information from each layer of the neural network.
Let f denote the objective function we wish to minimise. The Gauss-Newton matrix G of f is a common approximation of the Hessian H of f defined by

$$G = \nabla f \nabla f^T. \tag{18}$$

Note that this is sometimes written in the form $G = J^T J$, where $J = \nabla f^T$ is the Jacobian of f. The Fisher information matrix F is defined as the expected value of G

$$F = \mathbb{E}(G) \approx \mathbb{E}(H). \tag{19}$$

Given a neural network with l layers, F will be a block diagonal matrix with $l \times l$ blocks. Each block $\widetilde{F}_{ij}$ of F is given by

$$\mathbb{E}(\nabla_{w^i} f \otimes \nabla_{w^j} f) \tag{20}$$

where i and j are layers in the network and w^i the parameters in layer i. Each block is then approximated by

$$\mathbb{E}(\nabla_{w^i} f \otimes \nabla_{w^j} f) \approx \mathbb{E}(\widetilde{a}^{i-1}(\widetilde{a}^{j-1})^T) \otimes \mathbb{E}(\delta^i(\delta^j)^T) \tag{21}$$

where $\widetilde{a}^i$ is the activation values in layer i with an appended 1 in the last position of the vector i.e. $(\widetilde{a}^i)^T = [(a^i)^T 1]$. This is done as we are treating the weights and biases of the network together. The terms δ^i arise from standard backpropogation formulas.

The K-FAC algorithm uses a sparse representation of F and therefore only takes into account the diagonal blocks of F and sets the other blocks to zero. The diagonal blocks $\mathbb{E}(\widetilde{a}^{i-1}(\widetilde{a}^{i-1})^T) \otimes \mathbb{E}(\delta^i(\delta^i)^T)$ are called the Kronecker factors. In general, updates in the Newton method involve the inverse Hessian. From the property $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$ of the Kronecker product, we see that we can invert the approximation of the Fisher information by simply inverting the diagonal blocks of F. By abusing notation we shall denote the approximation to the Fisher information matrix, given by (21), by F.

The Gauss-Newton matrix is always positive semi-definite and this holds for F as well. However, in doing a Newton type update it is desirable for the Hessian approximation matrix to always be positive definite so as to avoid saddle points. Therefore, a damping factor of γ is implemented so that the Hessian approximation becomes $F + \gamma I$. The update for the K-FAC algorithm is then given by

$$x_{k+1} = x_k - \eta(F + \gamma I)^{-1}\nabla_{x_k} f \tag{22}$$

where η is a pre-chosen learning rate. For more details on K-FAC and the pseudocode for the algorithm the reader is referred to [4].

2. Theoretical Analysis

2.1. Analyzing the Hessian of a Coordinate Network

In this section we give details on matrix representations of the differential and Hessian of the MSE loss of a neural network, trained with a fixed data set. We then use these representations to give a proof of lemma 4.1 and proposition 4.2 from the paper.

2.1.1 Preliminaries

We start by setting up the notation we will be using. In general a coordinate MLP with L layers is formulated as a function $f : \mathbb{R}^{n_0} \to \mathbb{R}^{n_L}$ defined by

$$f : x \to T_L \circ \psi \circ T_{L-1} \circ \cdots \circ \psi \circ T_1(x) \tag{23}$$

where $T_i : x_i \to A_i x_i + b_i$ is an affine transformation with trainable parameters $A_i \in \mathbb{R}^{n_i \times n_{i_i}}$, $b_i \in \mathbb{R}^{n_i}$, and ψ_i is a non-linear activation acting component wise. The layer widths of the network are given by the numbers $\{n_1, n_2, \ldots, n_L\}$. The number n_0 is the input dimension and the number n_L is the output dimension. The composition $\psi \circ T_k \circ \cdots \circ \psi \circ T_1$ gives the first k-layers of the neural network function.

In the coming discussion we will need to treat our weights and biases as one parameter vector. We do this as follows: The input x_i will be redefined to the column vector $\bar{x}_i = (x_i^T, 1)^T$. We will also define

$$\bar{A}_i = \begin{bmatrix} A_i & b_i \\ 0 & 1 \end{bmatrix} \tag{24}$$

The affine transformation T_i will then be redefined to

$$T_i(\bar{x}_i) = \bar{A}_i \bar{x}_i = \begin{bmatrix} A_i & b_i \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ 1 \end{bmatrix} = \begin{bmatrix} A_i x_i + b_i \\ 1 \end{bmatrix} \tag{25}$$

and the non-linearity will only act on all but the last component, thereby giving

$$\psi \circ T_i(x_i) = \begin{bmatrix} \psi(A_i x_i + b_i) \\ 1 \end{bmatrix} \tag{26}$$

In this way we can treat the weights and biases as one parameter vector via equation (24). This does mean that to an input data vector, we will have to adjoin a 1 after the last row, and it also means that our output will have one dimension more, with the last value always being a 1. Furthermore, the size of the matrix in equation (24) is $(n_i + 1) \times (n_{i-1} + 1)$ but strictly

speaking only $(n_i+1)\times n_{i-1}$ entries are trainable. This will not create any problems as in such a case differentiating with respect to any of the non-trainable parameters will give zero, which will not affect the results of this section.
We will also fix our data sets once in for all. We assume we have a fixed data set $(X,Y)\in\mathbb{R}^{n_0\times N}\times\mathbb{R}^{n_L\times N}$ which consists of input data $X\in\mathbb{R}^{n_0\times N}$ and $Y\in\mathbb{R}^{n_L\times N}$ the targets. X will be thought as having feature dimension n_0 and the number of training samples will be given by N. Thus we can think of (X,Y) as a collection of training samples $\{(x_i,y_i\}_{i=1}^N$, with each pair (x_i,y_i) a training point.
Using the above notation we can view a coordinate MLP as a map

$$f:\mathbb{R}^{n_0\times N}\times\mathbb{R}^p\to\mathbb{R}^{n_L\times N} \tag{27}$$

where p denotes the parameter dimension. In general, a parameter vector $\theta\in\mathbb{R}^p$ will be written as $\theta=(\theta(1),\ldots,\theta(L))$, where each $\theta(i)\in\mathbb{R}^{n_i\times n_{i-1}}$ corresponds to the parameters of the ith-layer, given by equation (24). Thus from now on we combine the weights and biases as one parameter vector and assume that vector lives in $\mathbb{R}^{n_i\times n_{i-1}}$. Each such parameter $\theta(i)$ represents a matrix of trainable weights given by:

$$\theta(i)=\begin{bmatrix}\theta(i)_{11} & \cdots & \theta(i)_{1n_{i-1}}\\ \vdots & \vdots & \vdots\\ \theta(i)_{n_i1} & \cdots & \theta(i)_{n_in_{i-1}}.\end{bmatrix}$$

When we speak of the parameters of the ith-layer as a vector we will need to flatten such matrices. We will do so by flattening each row to a column, thereby obtaining a column vector. Thus the above $\theta(i)$ will be flattened to the vector

$$Vec(\theta(i))=(\theta(i)_{11},\ldots,\theta(i)_{1n_{i-1}},\ldots,\theta(i)_{n_i1},\ldots,\theta(i)_{n_in_{i-1}})^T. \tag{28}$$

We will always assume such flattening of matrices has been done and will not explicitly use the vec notation. At times we will have to go back and forth between the actual parameter matrix and its associated flattened vector. The context should make it clear as to which representation we are dealing with.
In order to write the network f as a composition of its layers we will write f in the following form:

$$f=f_L\circ\cdots\circ f_1 \tag{29}$$

where for each $1\le k\le L$

$$f_k:\mathbb{R}^{n_{k-1}\times N}\times\mathbb{R}^{n_k\times n_{k-1}}\times\cdots\times\mathbb{R}^{n_L\times n_{L-1}}\to\mathbb{R}^{n_k\times N}\times\mathbb{R}^{n_{k+1}\times n_k}\times\cdots\times\mathbb{R}^{n_L\times n_{L-1}} \tag{30}$$

is the map defined by

$$f_k(Z,\theta(k),\ldots,\theta(L))=(\psi(\theta(k)\cdot Z),\theta(k+1),\ldots,\theta(L))^T. \tag{31}$$

The quantity $\theta(k)\cdot Z$ is the matrix in $\mathbb{R}^{n_k\times N}$ given by applying $\theta(k)$ viewed as a $n_k\times n_{k-1}$ matrix acting on a $n_{k-1}\times N$ matrix Z. Furthermore, the latter entries $(\theta(k+1),\ldots,\theta(L))$ are all viewed as flattened vectors. This is an example of how we have had to use parameters both in their matrix form and their flattened vector form.
Using (31), the map f_1 is a map

$$f_1:\mathbb{R}^{n_0\times N}\times\mathbb{R}^{n_1\times n_0}\times\cdots\times\mathbb{R}^{n_L\times n_{L-1}}\to\mathbb{R}^{n_1\times N}\times\mathbb{R}^{n_2\times n_1}\times\cdots\times\mathbb{R}^{n_L\times n_{L-1}} \tag{32}$$

where $\mathbb{R}^{n_0}$ is the input space, the space in which our input data resides. As we will not be taking any derivatives with respect to data, and our data set has already been fixed, we will make life easier by viewing

$$f_1:\mathbb{R}^{n_1\times n_0}\times\cdots\times\mathbb{R}^{n_L\times n_{L-1}}\to\mathbb{R}^{n_1\times N}\times\mathbb{R}^{n_2\times n_1}\times\cdots\times\mathbb{R}^{n_L\times n_{L-1}} \tag{33}$$

defined by

$$f_1(\theta(1),\ldots,\theta(L))=(\psi(\theta(1)\cdot X),\theta(2),\ldots,\theta(L))^T \tag{34}$$

where X is our fixed input data in $\mathbb{R}^{n_0\times N}$.
One final notation we introduce is the following. Given a matrix $A\in\mathbb{R}^{m\times n}$, viewed as a $m\times n$ matrix, we let A^j denote the jth-row of A and A_j denote the jth column of A. With this notation, we observe the following: Given a parameter vector $\theta(k)\in\mathbb{R}^{n_k\times n_{k-1}}$, viewed as a $n_k\times n_{k-1}$ matrix, and a $Z\in\mathbb{R}^{n_{k-1}\times N}$. The product $\theta(k)\cdot Z$ is flattened to the vector

$$(\theta^1(k)\cdot Z_1,\ldots,\theta^1(k)\cdot Z_N,\ldots,\theta^{n_k}\cdot Z_1,\ldots,\theta^{n_k}\cdot Z_N)^T$$

where each $\theta^j(k)$ is a row vector with n_{k-1} entries and each Z_j is a column vector with n_{k-1} entries.

2.1.2 Proof of lemma 4.1

In this section we will primarily be concerned with derivatives of the MSE loss function with respect to parameters. We will write the MSE loss function as:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}|f(x_i,\theta) - y_i|^2. \tag{35}$$

Observe that the MSE loss can be written as the composition $c \circ f$, where c is a convex cost function given by the average squared error:

$$c : \mathbb{R}^{n_L \times N} \to \mathbb{R} \tag{36}$$

defined by

$$c(z_1, \ldots, z_N) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}|z_i - y_i|^2 \tag{37}$$

where we remind the reader that our data set consists of point $\{(x_i, y_i\}_{i=1}^{N}$, with each $x_i \in \mathbb{R}^{n_0}$ and $y_i \in \mathbb{R}^{n_L}$.
With this notation we can easily see that the MSE loss is given by the composition $c \circ f$. We now compute the differential of the MSE loss function, from which a simple transpose gives the gradient. Observe that the loss function is a map

$$\mathcal{L} : \mathbb{R}^p \to \mathbb{R} \tag{38}$$

and therefore its differential is a map

$$D\mathcal{L} : \mathbb{R}^p \to Lin(\mathbb{R}^p, \mathbb{R}) \cong \mathbb{R}^{p\times 1} \tag{39}$$

where $Lin(\mathbb{R}^p, \mathbb{R})$ denotes the linear maps from $\mathbb{R}^p$ to $\mathbb{R}$, which is linearly isomorphic to the space of $1 \times p$ matrices which we can identify as $\mathbb{R}^{p\times 1}$.
The chain rule gives $D\mathcal{L}(\theta) = Dc(f(\theta)) \cdot Df(\theta)$. Therefore, in order to compute the differential of the loss, it suffices to compute the differential of the cost function c and the differential of the neural network function f.
The following proposition is an easy consequence of equation (37).

Proposition 2.1. $Dc(Z) = \frac{1}{N}(Z - Y)$, *where Y denotes the matrix of targets from our data set.*

The next step is to compute the differential of the neural network Df. As we have already fixed our data set, and the differential is taken with respect to parameters, the neural network function is a map

$$f : \mathbb{R}^p \to \mathbb{R}^{n_L \times N}. \tag{40}$$

In order to compute the differential, we have to flatten the network f, which we do to a $n_L N \times 1$ column vector. The differential will then be a map

$$Df : \mathbb{R}^p \to Lin(\mathbb{R}^p, \mathbb{R}^{n_L N}) \cong \mathbb{R}^{p\times n_L N}. \tag{41}$$

By equation (29), and the chain rule, we have that for a vector $\theta \in \mathbb{R}^p$

$$Df(\theta) = Df_L(f_{L-1} \circ \cdots \circ f_1(\theta)) \cdot Df_{L-1}(f_{L-2} \circ \cdots \circ f_1(\theta)) \cdots Df_2(f_1(\theta)) \cdot Df_1(\theta). \tag{42}$$

We thus see that in order to compute the differential of the network f, it suffices by (42) to compute the differential of each f_k. We will introduce some notation that will help us write the differential Df_k in a convenient way.
We remind the reader that

$$f_k : \mathbb{R}^{n_{k-1}\times N} \times \mathbb{R}^{n_k \times n_{k-1}} \times \cdots \times \mathbb{R}^{n_L \times n_{L-1}} \to \mathbb{R}^{n_k \times N} \times \mathbb{R}^{n_{k+1}\times n_k} \times \cdots \times \mathbb{R}^{n_L \times n_{L-1}}$$

defined by

$$f_k(Z, \theta(k), \ldots, \theta(L)) = (\psi(\theta(k) \cdot Z), \theta(k+1), \ldots, \theta(L))^T.$$

see (31). The element $\psi(\theta(k) \cdot Z)$ is a matrix that has been flattened and can be expressed in the following way

$$\psi(\theta(k) \cdot Z) = (\psi(\theta^1(k) \cdot Z_1), \ldots, \psi(\theta^1(k) \cdot Z_N), \ldots, \psi(\theta^{n_k}(k) \cdot Z_1), \ldots, \psi(\theta^{n_k}(k) \cdot Z_N))^T.$$

We then define

$$\psi(\theta^j(k)\cdot Z) = (\psi(\theta^j(k)\cdot Z_1),\dots,\psi(\theta^j(k)\cdot Z_N))^T.$$

We use the following notation to denote partial derivatives with respect to the space variable Z and the parameter variable $\theta(j)$, for $k \le j \le L$. The partial derivative $\frac{\partial}{\partial Z}$ will denote derivatives with respect to the variable Z, and $\frac{\partial}{\partial \theta^i(j)}$ will denote derivatives with respect to the ith row of the parameter variable $\theta(j)$ for $k \le j \le L$. Thus for example, we have that

$$\frac{\partial}{\partial Z}\psi(\theta^j(k)\cdot Z) = \left(\frac{\partial}{\partial Z}\psi(\theta^j(k)\cdot Z_1),\dots,\frac{\partial}{\partial Z}\psi(\theta^j(k)\cdot Z_N)\right)^T. \tag{43}$$

We have the following simple lemma

Lemma 2.2. *$\frac{\partial}{\partial Z_j}\psi(\theta^i(k)Z_i) = 0$ for all $j \neq i$ and $\frac{\partial}{\partial \theta^j(k)}\psi(\theta^i(k)Z_i) = 0$ for all for all $j \neq i$.*

Proof. The result follows immediately from noting that the term $\psi(\theta^i(k)Z_i)$ does not depend on the variable Z_j and $\theta^j(k)$ for $j \neq i$. □

We now give a matrix formula for the differential of f_k.

Proposition 2.3. *The differential Df_k is a $(Nn_k + n_{k+1}n_k \cdots + n_L n_{L-1}) \times (n_{k-1}N + n_k n_{k-1} + \cdots + n_L n_{L-1})$ matrix given by the following representation*

$$\begin{bmatrix}
\frac{\partial}{\partial Z}\psi(\theta^1(k)\cdot Z) & \frac{\partial}{\partial \theta^1(k)}\psi(\theta^1(k)\cdot Z) & 0 & \cdots\cdots & 0 & 0 & \cdots & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \\
\frac{\partial}{\partial Z}\psi(\theta^{n_k}(k)\cdot Z) & 0 & 0 & \cdots\cdots & \frac{\partial}{\partial \theta^{n_k}(k)}\psi(\theta^{n_k}(k)\cdot Z) & 0 & \cdots & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & I_{n_{k+1}n_k} & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & I_{n_L n_{L-1}}
\end{bmatrix}$$

where I_i denotes an $i \times i$ identity matrix.

Proof. The matrix representation follows from a straight forward calculation of partial derivatives which we explain. First of all we remind the reader that the image of f_k is to be thought of as a flattened column vector, where remember that we flatten each row to a column. By definition

$$f_k(Z,\theta(k),\dots,\theta(L)) = (\psi(\theta(k)\cdot Z),\theta(k+1),\dots,\theta(L))^T.$$

We now observe that the terms $\theta(k+1),\dots,\theta(L)$ will give zero when we apply $\frac{\partial}{\partial Z}$ and $\frac{\partial}{\partial \theta^i(k)}$ for $1 \le i \le n_k$. This leads to the zeros in the bottom left of the big matrix. The term $\psi(\theta(k)\cdot Z)$ will give zero when we apply the derivatives $\frac{\partial}{\partial \theta(k+1)},\dots,\frac{\partial}{\partial \theta(L)}$. This leads to the zeros in the top left of the big matrix.
The matrix representation of Df_k now follows from the derivatives $\frac{\partial}{\partial Z}\psi(\theta^i(k)\cdot Z)$ and $\frac{\partial}{\partial \theta(k)}\psi(\theta^i(k)\cdot Z)$, noting that by lemma 2.2 we have that $\frac{\partial}{\partial^j \theta(k)}\psi(\theta^i(k)\cdot Z) = 0$ for $i \neq j$. □

The proof of lemma 4.1 in the paper now follows by applying equation (42) and proposition 2.3.

2.1.3 Proof of proposition 4.2

The Hessian of the loss function $\mathcal{L}$ can be computed by applying the fact that $\mathcal{L} = c \circ f$ and the chain rule. We will use the notation $D^2\mathcal{L}$ to denote the Hessian of the loss, which is to be thought of as the second differential of $\mathcal{L}$.

Proposition 2.4. *Given a point $\theta \in \mathbb{R}^p$ we have that*

$$D^2\mathcal{L}(\theta) = Df(\theta)^T \cdot (\frac{1}{N}I)\cdot Df(\theta) + \frac{1}{N}(f(\theta) - Y)\cdot D^2 f \tag{44}$$

where $\frac{1}{N}I$ denotes the identity matrix with $1/N$ on its diagonal and recall that Y is a matrix consisting of the targets from the fixed data set.

Proof. This follows by applying the chain and product rule to $D\mathcal{L}$. Given a point θ, we have that

$$D\mathcal{L}(\theta) = Dc(f(\theta)) \cdot Df(\theta) \tag{45}$$

Differentiating once more, and applying the chain and product rules, we get

$$D^2\mathcal{L}(\theta) = Df(\theta)^T \cdot D^2c(f(\theta)) \cdot Df(\theta) + Dc(f(\theta)) \cdot D^2f(\theta). \tag{46}$$

By proposition 2.1, we have that $Dc(f(\theta)) = \frac{1}{N}(f(\theta) - Y)$ and then differentiating once more we get $D^2c(f(\theta) = \frac{1}{N}I$ and the result follows. $\square$

Proposition 2.4 implies that in order to compute the Hessian of the loss, we need to compute the Hessian of the neural network function.

Lemma 2.5. *We have the following decomposition for the Hessian of f*

$$\begin{aligned}
D^2f = &(Df_1)^T \cdots (Df_{L-1})^T D^2 f_L (Df_{L-1}) \cdots (Df_1) \\
&+ (Df_1)^T \cdots (Df_{L-2})^T Df_L D^2 f_{L-1} (Df_{L-2}) \cdots (Df_1) \\
&+ (Df_1)^T \cdots (Df_{L-3})^T Df_L Df_{L-1} D^2 f_{L-2} (Df_{L-3}) \cdots (Df_1) \\
&+ \cdots + (Df_1)^T Df_L Df_{L-1} \cdots Df_3 D^2 f_2 D^2 f_2 (Df_1) \\
&+ Df_L Df_{L-1} \cdots Df_2 D^2 f_1
\end{aligned}$$

Proof. The proof of this follows by induction on the layers. $\square$

Lemma 2.5 implies that in order to compute the Hessian of the neural network, we need to compute the Hessian and the differential of each layer. The differential of each of the layers was already computed in proposition 2.3. We will now give a matrix formula for the Hessian of each layer f_k.
In order to compute the Hessian D^2f_k, we will flatten Df_k so that it is a map

$$Df_k : \mathbb{R}^{n_{k-1}\times N} \times \mathbb{R}^{n_k \times n_{k-1}} \times \cdots \times \mathbb{R}^{n_L \times n_{L-1}} \to \mathbb{R}^{(n_k\times N + n_{k+1}\times n_k + \cdots + n_L \times n_{L-1})(n_{k-1}\times N + n_k \times n_{k-1} + \cdots + n_L \times n_{L-1})}. \tag{47}$$

Then for a point $(Z, \theta(k), \ldots, \theta(L))$, we have that $D^2f_k(Z, \theta(k), \ldots, \theta(L))$ will be a $((n_k \times N + n_{k+1} \times n_k + \cdots + n_L \times n_{L-1})(n_{k-1} \times N + n_k \times n_{k-1} + \cdots + n_L \times n_{L-1})) \times (n_{k-1} \times N + n_k \times n_{k-1} + \cdots + n_L \times n_{L-1})$ matrix. In fact, it can be thought of as a collection of $(n_k \times N + n_{k+1} \times n_k + \cdots + n_L \times n_{L-1})$ square $(n_{k-1} \times N + n_k \times n_{k-1} + \cdots + n_L \times n_{L-1}) \times (n_{k-1} \times N + n_k \times n_{k-1} + \cdots + n_L \times n_{L-1})$ matrices stacked on top of each other.
Each such square matrix arises from the rows of the matrix representation of Df_k, see proposition 2.3. We start by computing these square matrices.

Lemma 2.6. *Given the matrix representation of Df_k in proposition 2.3 and $1 \leq i \leq n_k N$, we have that the derivative of the ith-row of Df_k is given by*

$$\begin{bmatrix}
\frac{\partial^2}{\partial Z \partial Z}\psi(\theta^i(k)\cdot Z) & 0 & \cdots & 0 & \frac{\partial^2}{\partial \theta^i(k)\partial Z}\psi(\theta^i(k)\cdot Z) & 0 & \cdots & 0 \\
0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
\frac{\partial^2}{\partial \partial Z \theta^i(k)}\psi(\theta^i(k)\cdot Z) & 0 & \cdots & 0 & \frac{\partial^2}{\partial \theta^i(k)\partial \theta^i(k)}\psi(\theta^i(k)\cdot Z) & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0
\end{bmatrix}$$

Furthermore if $n_k N < i \leq n_{k+1} \times n_k + \cdots + n_L \times n_{L-1}$ then the derivative of the ith-row of Df_k will be a matrix of zeros.

Proof. The proof follows by inspecting each row of the matrix representation of Df_k given in proposition 2.3. We observe that if $1 \leq i \leq n_k N$, then the ith row of Df_k is given by

$$\begin{bmatrix} \frac{\partial}{\partial Z}\psi(\theta^i(k)\cdot Z) & 0 & 0 & \cdots\cdots & 0 & \frac{\partial}{\partial \theta^i(k)}\psi(\theta^i(k)\cdot Z) & 0 & \cdots & \cdots & 0 \end{bmatrix}$$

We then observe that the derivatives $\frac{\partial}{\partial \theta^j(k)}$ of any element in the above row will be zero for $j \neq i$ as none of the elements in the row depend on the variable $\theta^j(k)$ when $j \neq i$. This means the only derivatives that could possibly be non-zero for such a row will come from $\frac{\partial}{\partial Z}$ and $\frac{\partial}{\partial \theta^i(k)}$. This proves the first part of the proposition. To prove the second part, we simply observe that the ith-rows of Df_k for $n_k N < i \leq n_{k+1} \times n_k + \cdots + n_L \times n_{L-1}$ have only one non-zero entry which will be a 1. When differentiated with respect to any of the variables this will give zero, and thus we simply get the zero matrix for such a row. This proves the second part of the proposition. ◻

Using lemma 2.6, we can compute a full matrix representation of the Hessian of f_k.

Proposition 2.7. *A matrix representation of the Hessian of f_k is given by*

$$\begin{bmatrix}
\frac{\partial^2}{\partial Z \partial Z}\psi(\theta^1(k)\cdot Z) & \frac{\partial^2}{\partial \theta^1(k)\partial Z}\psi(\theta^1(k)\cdot Z) & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
\frac{\partial^2}{\partial Z \partial \theta^1(k)}\psi(\theta^1(k)\cdot Z) & \frac{\partial^2}{\partial \theta^1(k)\partial \theta^1(k)}\psi(\theta^1(k)\cdot Z) & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\
\frac{\partial^2}{\partial Z \partial \theta^{n_k}(k)}\psi(\theta^{n_k}(k)\cdot Z) & 0 & 0 & \cdots & 0 & \frac{\partial^2}{\partial \theta^{n_k}(k)\partial \theta^{n_k}(k)}\psi(\theta^{n_k}(k)\cdot Z) & 0 & \cdots & 0 \\
0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\
0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
\frac{\partial^2}{\partial Z \partial \theta^{n_k}(k)}\psi(\theta^{n_k}(k)\cdot Z) & 0 & 0 & \cdots & 0 & \frac{\partial^2}{\partial \theta^{n_k}(k)\partial \theta^{n_k}(k)}\psi(\theta^{n_k}(k)\cdot Z) & 0 & \cdots & 0 \\
0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\
0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
\vdots & & & & & & & &
\end{bmatrix}$$

Proof. The proof follows by taking each square matrix from lemma 2.6 and stacking them on top of each other. ◻

The goal of this section is to prove proposition 4.2 from the paper, which analyses the Hessian of a ReLU activated coordinate network. In order to do this we will need to employ the theory of distributions. The reader who is not familiar with this theory can consult the references [6], [7].

We will prove that derivatives of the ReLU function can be interpreted as distributions.

Proposition 2.8. *1. $\frac{d}{dx}ReLU(x) = H(x)$, where H is the step function centered at 0.*

2. Viewing $\frac{d}{dx}ReLU(x)$ as a distribution, we have that $\frac{d^2}{dx^2}ReLU = \delta$, where δ denotes a Dirac delta distribution centered at 0.

Proof. By definition $ReLU(x) = max(0, x)$, therefore for $x < 0$ is is clear that $\frac{d}{dx}ReLU(x) = 0$. For $x \geq 0$, we have that $ReLU(x) = x$ and therefore $\frac{d}{dx}ReLU(x) = 1$ for such points. It follows that one can represent the derivative $\frac{d}{dx}ReLU(x) = H(x)$ distributionally, with a discontinuity at the origin.

We move on to proving the second identity. Given a function $f \in C_c^\infty(\mathbb{R})$ the distribution H is defined by

$$\langle H, f\rangle = \int_{\infty}^{\infty} H(x)f(x)dx = \int_0^\infty f(x)dx.$$

The derivative of H is then given by (see [6] for preliminaries on derivatives of distributions)

$$\langle H', f\rangle = -\langle H, f'\rangle = -\int_0^\infty f'(x)dx = f(0) = \langle \delta, f\rangle$$

where the second equality comes from the fundamental theorem of calculus and the fact that f is compactly supported. The final equality follows by definition of the Dirac delta distribution. It thus follows that $\frac{d^2}{dx^2}ReLU = \delta$ as distributions. □

We can now make a few observations. Given a ReLU activated coordinate network f, proposition 2.4 tells us that the Hessian of the loss of such a network has two main components. The first given by first order derivatives of the network and the second given by second order derivatives of the network. Lemma 2.5 shows us that by the chain rule, the Hessian of f is determined by the Hessian of each layer f_k together with various of derivatives of the layers f_l for $l \neq k$. We can now prove proposition 4.2 from the paper.

Proof of proposition 4.2: From the above discussion we have that the Hessian of a ReLU activated coordinate network f can be analysed by the derivatives and Hessian of each component layer f_k. From proposition 2.7 and proposition 2.8 each component of the Hessian of f_k will contain a scaled Dirac delta distribution, centered possibly away from zero. Furthermore, from proposition 2.3 and proposition 2.8 we see that each component of Df_k is a step function, possibly centered away from zero. Finally, applying lemma 2.5 and proposition 2.4 we see that the Hessian of the loss of such a network will have terms consisting of sums of Dirac deltas (possibly scaled and centered away from zero) and step functions (possibly centered away from zero). This finishes the proof.

Caveat. In the above proof we used lemma 2.5 to represent the Hessian of the network in terms of the Hessian of its layers. This representation involves matrix products of the Hessian with the differential of other layers. The differential will contain step function components and the Hessian will contain Dirac delta components. When doing a matrix multiplication these components will multiply together, producing terms that are products of step functions with Dirac delta terms. In the case that the centre of the step function and the delta distribution coincide, this product cannot be given the structure of a distribution. In general, the product of two distributions may not be a distribution, see [2], thus strictly speaking such products may not be well defined. We will not worry about this technicality as the next paragraph explains that we can approximate the Hessian of the layer by the zero matrix.
Since a Dirac delta distribution is almost everywhere zero, we see that the Hessian of a ReLU activated network is almost everywhere zero. Using proposition 2.4, we see that for a ReLU coordinate network f, the Hessian of the loss can be well approximated by $Df(\theta)^T Df(\theta)$. We now look at the the matrix representation in proposition 2.3 and make two important observations. The first is that every column but the first one containt only one possible non-zero element given by a parameter derivative or a 1 coming from an identity matrix. The second is that the first column contains Nn_k possible non-zero terms, all given by spatial derivatives with respect to Z. Thus we see that if one of the parameter derivatives of the form $\frac{\partial}{\partial \theta^i(k)} \psi(\theta^i(k) \cdot Z) = 0$, then the matrix will be rank deficient. If our network has coordinates drawn from a unit cube about the origin and our parameters are initialised via a distribution centered at the origin, then we see it is likely that one of the terms in Df_k will vanish for each layer, causing Df to be rank deficient, which in turn will cause the Hessian of the loss $\mathcal{L}$ to be rank deficient.
The above predictions motivate a key reason for using non-traditional activations such as Sine or Gaussian. Both these functions have derivatives that near zero either do not vanish or only vanish at the zero point. In contrast a ReLU function has first order derivative that vanishes for all negative numbers and has second (and higher) order derivatives that vanish almost everywhere.

2.2. Analyzing L-BFGS on a Coordinate Network

2.2.1 Proofs of Theorem 4.4 and 4.5

In this section, we give the proofs of theorems 4.4 and 4.5 from the main paper.

Proof of theorem 4.4: To prove 1, we observe that by Prop. 4.2 (from the main paper) there will be points of parameter space where the Hessian will contain terms consisting of step functions and Dirac delta functions. Such terms are not continuous. By the chain rule it follows that $\mathcal{L}$ is not twice continuously differentiable at every parameter point.
To prove 2, we observe that at a non-continuously differentiable minimum θ^*, the Hessian of $\mathcal{L}$ at θ^* will not be continuous at θ^* by Prop. 4.2 (from the main paper). Hence it cannot be Lipshitz continuous locally about θ^*. By Thm. 4.3 (from the main paper), it follows that superlinear convergence to θ^* cannot be guaranteed.

Proof of theorem 4.5: To prove 1, we simply observe that a sine or Gaussian functions is twice continuously differentiable. This implies that a sine-/Gaussian-activated coordinate network is twice continuously differentiable via the chain rule. Another application of the chain rule, see the definition of $\mathcal{L}$ given in eqn. (2) from the main paper, implies 1 holds.
To prove 2, we first observe that since the second derivatives of a sine or Gaussian function is Lipshitz continuous, the chain rule, and the formula for $\mathcal{L}$, see eqn. (2) from main paper, implies that the Hessian H of $\mathcal{L}$ is Lipshitz continuous about a neighbourhood of θ^*. Applying Thm. 4.3 (from the main paper) we find that the convergence rate is superlinear

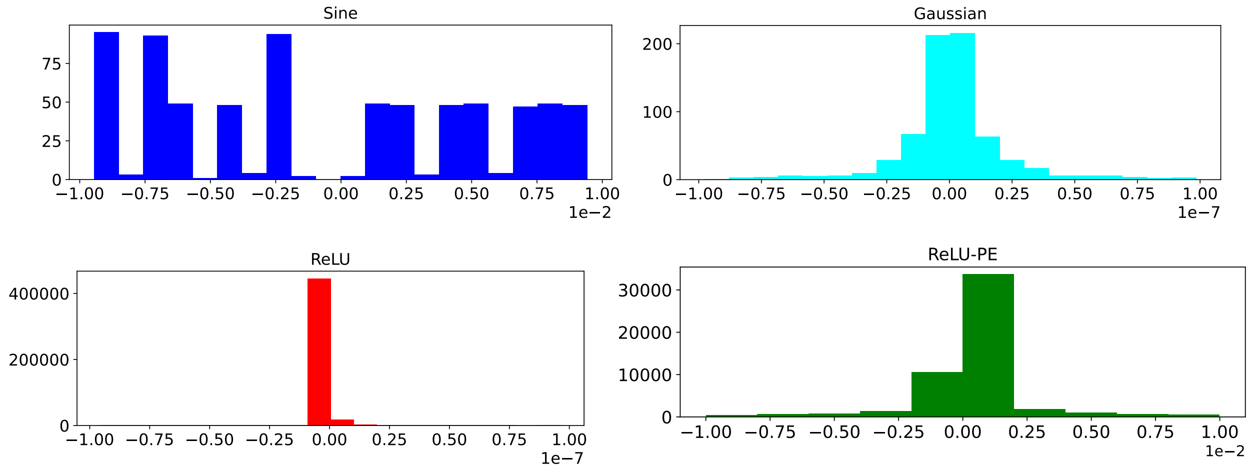


Figure 1: Total eigenvalue distribution of the Hessian of MSE loss for different networks, each with **4 hidden layers and 16 neurons**, throughout training. ReLU- and ReLU-PE-activated network has 28% and 45% of its eigenvalues at 0, respectively. In contrast, the smallest eigenvalue for Sine- and Gaussian-activated network is 5×10^{-4} and 1×10^{-12}, respectively. This highlights the superior conditioning of the Hessian of a sine- and Gaussian-activated network (**no** zero eigenvalues) compared to a ReLU one (**many** zero eigenvalues).

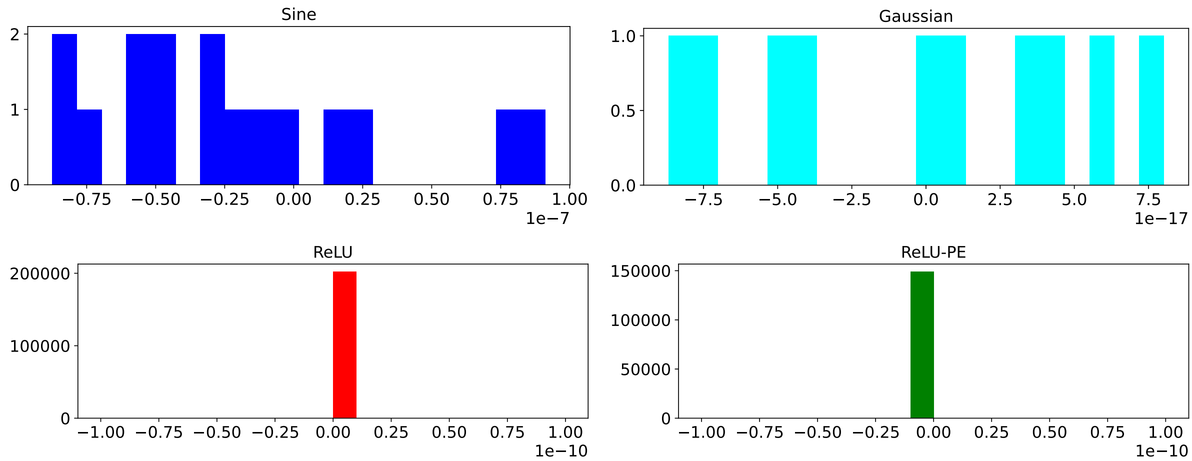


Figure 2: Total eigenvalue distribution of the Hessian of MSE loss for different networks, each with **4 hidden layers and 64 neurons**, throughout training. ReLU and ReLU-PE has has 31% and 21% of its eigenvalues at 0, respectively. On the other hand, the smallest eigenvalue for Sine- and Gaussian-activated network is 5×10^{-9} and 1×10^{-18}, respectively. This highlights the superior conditioning of the Hessian of a Sine- and Gaussian-activated network (**no** zero eigenvalues) compared to a ReLU one (**many** zero eigenvalues).

3. Analyzing the Hessian During Training

In this section, we give empirical results on the Hessian of the loss of a coordinate network, offering a more in-depth study of the results from Sec. 4.2 of the **main paper** and verifying empirically the predictions made from Sec. 4.1 of the **main paper** and Sec. 2.1 of this supplementary.

We ran experiments to determine the conditioning of the Hessian of the MSE loss of different activated coordinate networks. All networks were trained on a 50×50 image reconstruction task, with a full sampling scheme for 50 iterations. We computed the eigenvalues of the Hessian of the MSE loss at each iteration throughout training. Experiments were carried out for different depth, width and initialisation scheme. We observed with each experiment that the non-traditional activations always had the Hessian of the loss having no zero eigenvalues, while the traditional activated networks had Hessians that

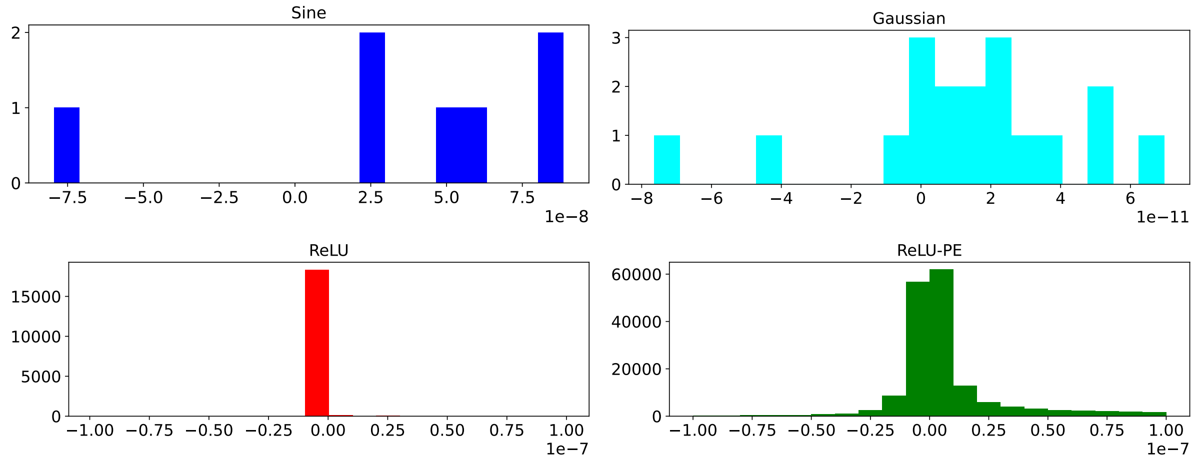


Figure 3: Total eigenvalue distribution of the Hessian of MSE loss for different networks, each with **1 hidden layers and 512 neurons**, throughout training. ReLU has 12% of its eigenvalues at 0, whereas ReLU-PE has the smallest eigenvalue of 3×10^{-14}. In contrast, the smallest eigenvalue for Sine- and Gaussian-activated network is 5×10^{-8} and 1×10^{-12}, respectively. This highlights the superior conditioning of the Hessian of a Sine- and Gaussian-activated network (**no** zero eigenvalues) compared to a ReLU one (**many** zero eigenvalues).

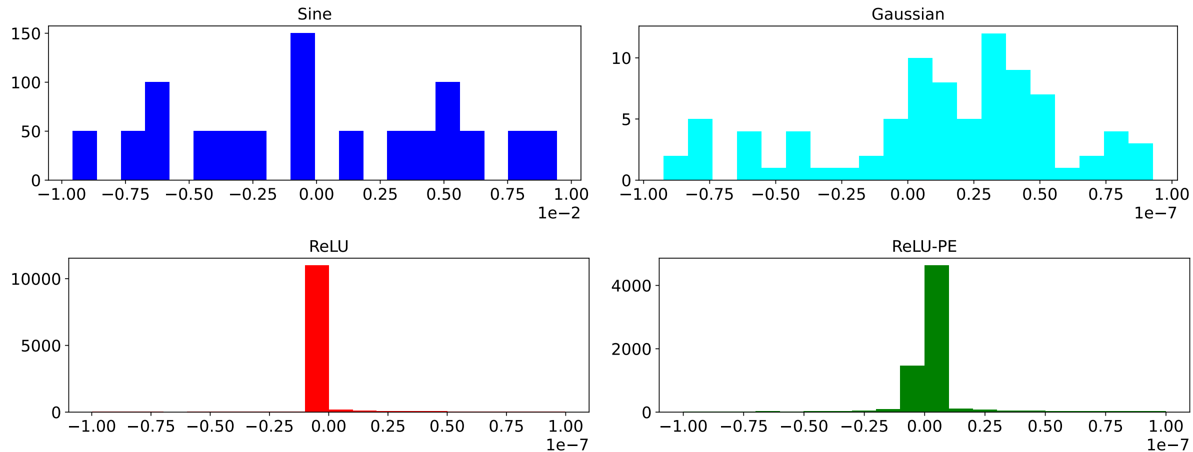


Figure 4: Total eigenvalue distribution of the Hessian of MSE loss for different networks, each with **4 hidden layers and 16 neurons**, initliased using **Xavier's normal** initialisation, throughout training. ReLU- and ReLU-PE-activated network has 24% and 5% of its eigenvalues at 0, respectively. In contrast, the smallest eigenvalue for Sine- and Gaussian-activated network is 1×10^{-4} and 1×10^{-12}, respectively. This highlights the superior conditioning of the Hessian of a Sine- and Gaussian-activated network (**no** zero eigenvalues) compared to a ReLU one (**many** zero eigenvalues).

admitted a large number of zero eigenvalues.
Fig. 1 shows the eigenvalues in a small neighbourhood of 0 of the Hessian of the loss of a 4-layer, 16-width neural network with 4 different activations, Sine, Gaussian, ReLU-PE, ReLU. From the figure, we see that both the Sine and Gaussian activated networks have no zero eigenvalues, showing that their loss functions have Hessians that are of full rank. In contrast, both the ReLU-PE and ReLU networks have a large number of zero eigenvalues, showing that their loss functions have Hessians that are rank deficient. For this experiment, the Sine activated network was initialised using Sirens initialisation scheme, while all other networks were initialised using a Kaiming uniform scheme.
Fig. 2 and Fig. 3 repeats the same experiment that was carried out above in the case of a 4-hidden layer, 64-width and 1-hidden layer, 512-width network, respectively. As can be seen from the figures, the non-traditional activations, Sine and Gaussian, have Hessians that have no zero eigenvalues, while the ReLU and ReLU-PE ones have a large number of zero eigenvalues.
Fig. 4 shows the results of the eigenvalues of the Hessian of the MSE loss of different activated MLPs, using a Xavier normal initialisation scheme. Similarly, we observe that the Sine and Gaussian activated MLPs have Hessians that have no zero eigenvalues, while the ReLU-PE and ReLU MLPs have a large number of zero eigenvalues.

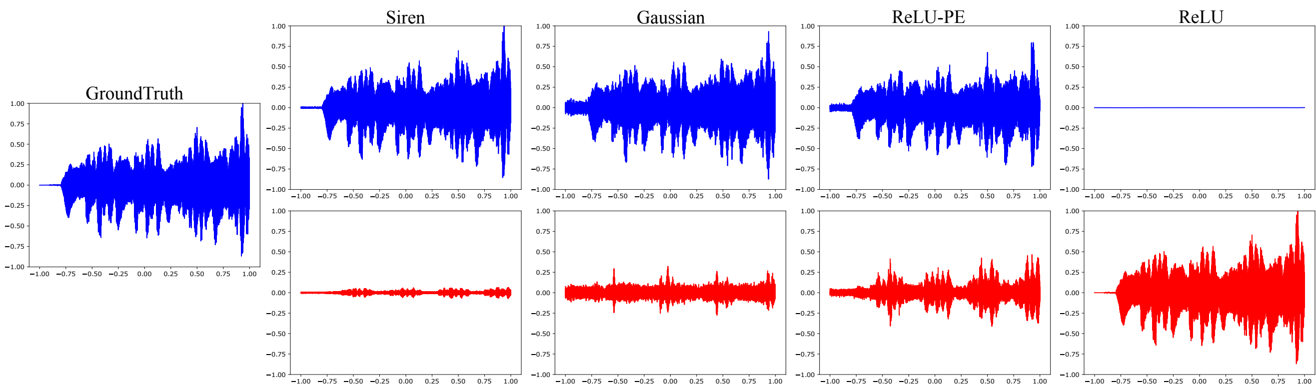


Figure 5: Fitted audio signals and error for various coordinate networks optimized using L-BFGS. We compare with architectures implemented with Sine, Gaussian, ReLU-PE ($L = 8$) as well as ReLU. *Top row*: Predicted signal. *Bottom row*: Error between the groundtruth waveform and the predicted waveform.

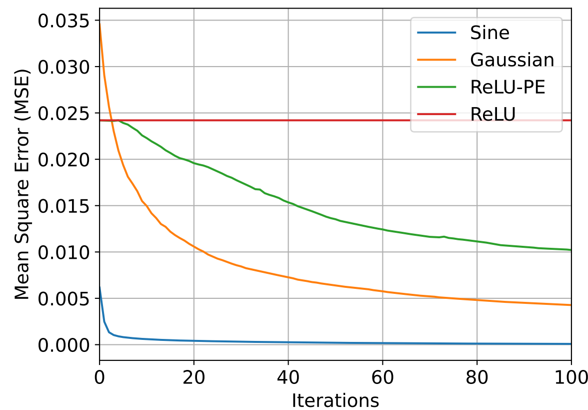
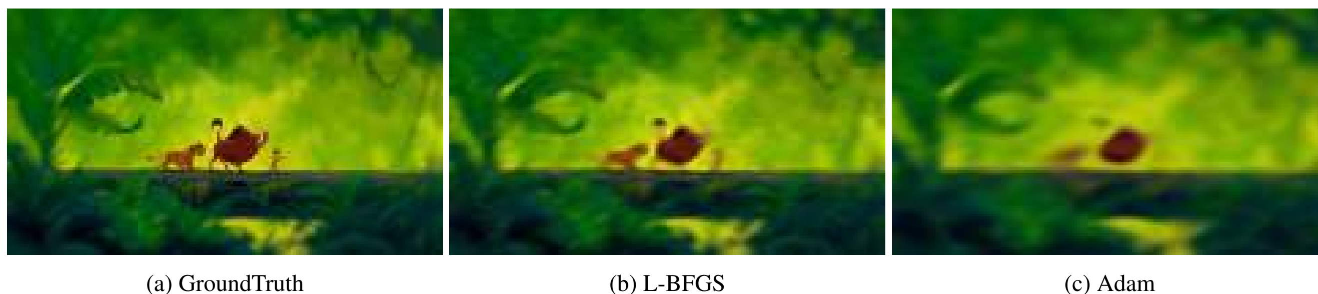


Figure 6: Training convergence for various coordinate networks optimized using L-BFGS on audio fitting. Sine-activated network achieves the fastest convergence speed compared to other activations with L-BFGS.



(a) GroundTruth (b) L-BFGS (c) Adam

Figure 7: **Video Reconstruction.** We provide the qualitative result on one snapshot of reconstructed frames from fitting the Lion King video. L-BFGS has achieved a substantially better reconstruction than Adam after trained for 500 iterations (31.68 vs. 28.05 dB) (L-BFGS vs. Adam). L-BFGS achieves convergence $\sim 2\times$ faster than Adam.

4. Additional Results of Experiments

4.1. Audio Reconstruction

We fit audio signals using various coordinate networks optimized using L-BFGS. We use the music data from Bach's Cello Suite No.1 (https://drive.google.com/drive/folders/1_iq__37-hw7FJOEUK1tX7mdp8SKB368K). We use a 4-hidden layers, 256 width MLP. For the Sine-activated network, we use frequency $\omega_o = 3000$ for the first two layers of the network and $\omega_o = 30$ for the rest of the layers of the network. We find that these frequencies generally work well to account for the high sampling of audio signals when Sine is optimized with L-BFGS. For the Gaussian activated network, we use sigma $\sigma = 0.01$.

Fig. 5 compares the reconstructed audio signals for various coordinate networks optimized using L-BFGS. Sine-activated coordinates produces an accurate waveform. Furthermore, the non-traditional activations (Sine and Gaussian) converge significantly faster than the traditional ones (ReLU and ReLU-PE), see Fig. 6. We refer the readers to see the supplemental folder for `demo_audio` containing the reconstructed audio signals.

4.2. Video Reconstruction

We fit a video using a Sine-MLP with 4-hidden layers and 64 neurons. We use the Lion King video available from https://www.youtube.com/watch?v=mzABW42AIhM&t=234s. We crop and downsample the video to 90×160 resolution. We use frequency $\omega_o = 30$ for all the layers of the network. Fig 7 shows a reconstruction snapshot of the video. Given the same amount of iterations, L-BFGS has achieved a substantially better PSNR (31.68dB) compared to Adam (28.05dB). We refer the readers to see the supplemental folder for `demo_video` containing the associated reconstructed video. Overall, L-BFGS achieves convergence $\sim 2\times$ faster than Adam.

4.3. 3D Shape Reconstruction

We fit a 3D shape using a Sine-MLP with 4-hidden layers and 64 neurons. Specifically, we aim to optimize a binary occupancy field, which represents a 3D shape as the decision boundary of a MLP [8, 3, 1]. We use the *bird* instance obtained from Thingi10K [9]. We sampled total of 100k points. Fig. 8 compares the qualitative result of the reconstructed mesh of L-BFGS and Adam within the same amount of training time of 3.5 seconds. Our results indicate the L-BFGS produced a complete reconstruction, while Adam was unable to do so. L-BFGS achieves convergence $\sim 2\times$ faster than Adam.

4.4. KiloImage

We provide result for more instances on KiloImage, see Fig. 9, 10, 11, 12, 13.

4.5. KiloVideo

KiloVideo is a collection of tiny independent MLPs trained on a high resolution video that has been decomposed into small grids. Each grid is represented using a small Sine-MLP, with 3 hidden layers and 64 neurons. 10k points are sampled on each grid. Here, we fit the 100 frames of Lion King video (https://www.youtube.com/watch?v=mzABW42AIhM&t=234s) (700×1200 resolution) by 8400 tiny Sine-activated coordinate networks. On average, L-BFGS achieves $\sim 4\times$

faster convergence than Adam, with some of the MLPs achieves $\sim 11\times$ faster than Adam. We refer the readers to see the supplemental folder for `demo_kilovideo` containing the reconstructed audio signals after training for 0.06 seconds.

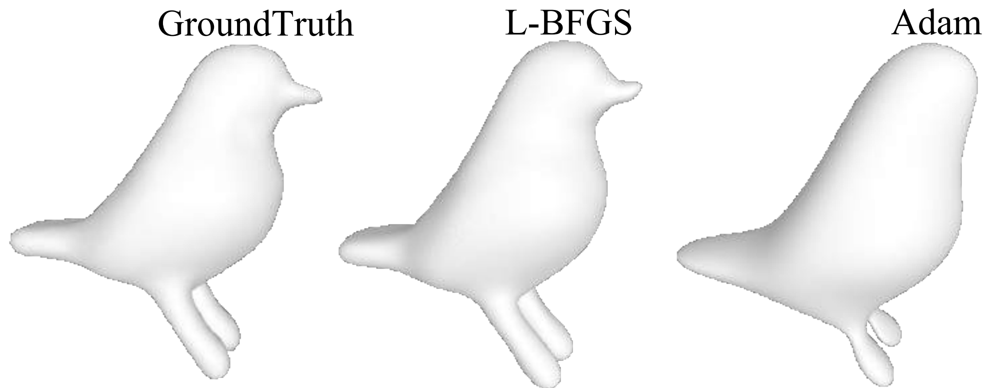


Figure 8: **3D Shape Reconstruction** on *bird* instance [9]. We provide the qualitative results of the reconstructed mesh. Given the same amount of training time, L-BFGS produced a complete reconstruction, while Adam was unable to do so. L-BFGS achieves convergence $\sim 2\times$ faster than Adam.

(a) L-BFGS

(b) Adam

Figure 9: **Gigapixel Image Reconstruction** on *A girl with a Pearl Earring Image* (4000×4000 resolution) by Johannes Vermeer (**CC BY 4.0**). Our approach represents this image using 1600 sine-activated MLPs. Using L-BFGS, our method achieves a substantially higher-fidelity reconstruction (34.41 dB) than Adam (29.71 dB), given the same amount of training time (average 0.8 seconds). L-BFGS achieves convergence (40.05 dB) $\sim 5\times$ faster than Adam.

(a) L-BFGS

(b) Adam

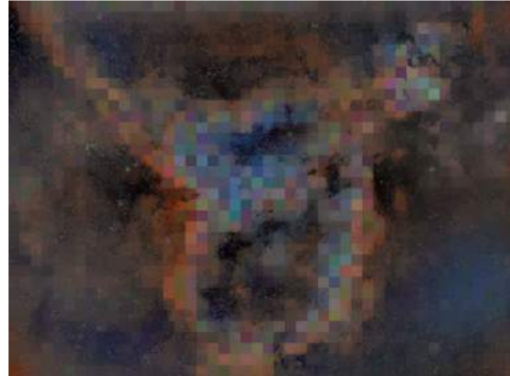
Figure 10: **Gigapixel Image Reconstruction** on *Wildlife Photograph of the Year 2019 National Geographic* (3000×3000 resolution) by Bao Yongqing (**CC BY 4.0**). Our approach represents this image using 900 sine-activated MLPs. Using L-BFGS, our method achieves a substantially higher-fidelity reconstruction (37.22 dB) than Adam (13.88 dB), given the same amount of training time (average 0.08 seconds). L-BFGS achieves convergence $\sim 6\times$ faster than Adam.

(a) L-BFGS

(b) Adam

Figure 11: **Gigapixel Image Reconstruction** on *Grindelwald* (1800×900 resolution) (**CC BY 4.0**). Our approach represents this image using 162 sine-activated tiny-MLPs. Using L-BFGS, our method achieves a substantially higher-fidelity reconstruction (34.41 dB) than Adam (29.71 dB), given the same amount of training time (average 0.09 seconds). L-BFGS achieves convergence (37dB) $\sim 6\times$ faster than Adam.

(a) L-BFGS



(b) Adam

Figure 12: **Gigapixel Image Reconstruction** on *Heart Nebula Image* (3400×4600 resolution) by Ram Samudrala (**CC BY 4.0**). Using L-BFGS, our method achieves a substantially higher-fidelity reconstruction (32.04 dB) than Adam (22.33 dB), given the same amount of training time (average 0.09 seconds). L-BFGS achieves convergence (40.03 dB) $\sim 5\times$ faster than Adam.

(a) L-BFGS

(b) Adam

Figure 13: **Gigapixel Image Reconstruction** on *The pillars of creation Image* (2000×1000 resolution) by NASA's James Webb Space Telescope). Using L-BFGS, our method achieves a substantially higher-fidelity reconstruction (23.48 dB) than Adam (18.05 dB), given the same amount of training time (average 0.09 seconds). L-BFGS achieves convergence (30dB) $\sim 4\times$ faster than Adam.

References

[1] Matan Atzmon and Yaron Lipman. Sal: Sign agnostic learning of shapes from raw data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2565–2574, 2020. 15

[2] Hajer Bahouri, Jean-Yves Chemin, and Raphaël Danchin. *Fourier analysis and nonlinear partial differential equations*, volume 343. Springer, 2011. 11

[3] Amos Gropp, Lior Yariv, Niv Haim, Matan Atzmon, and Yaron Lipman. Implicit geometric regularization for learning shapes. *ICML*, 2020. 15

[4] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015. 4, 5

[5] Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 1999. 1

[6] W Rudin. Functional analysis tata mcgraw, 1973. 10

[7] Robert S Strichartz. *A guide to distribution theory and Fourier transforms*. World Scientific Publishing Company, 2003. 10

[8] Peng-Shuai Wang, Yang Liu, Yu-Qi Yang, and Xin Tong. Spline positional encoding for learning 3d implicit signed distance fields. *arXiv preprint arXiv:2106.01553*, 2021. 15

[9] Qingnan Zhou and Alec Jacobson. Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797*, 2016. 15, 16