Supplementary Material for: Preconditioners for the Stochastic Training of Neural Fields

Shin-Fang Chng* Hemanth Saratchandran* Simon Lucey
Australian Institute of Machine Learning, University of Adelaide.

In this supplementary material, we provide the mathematical proofs for Theorems 4.2 and 4.3 from the main paper in Sec. 1, details of the KFAC and Shampoo algorithms in Sec. 2, additional results in Sec. 3, and the reproducibility details in Sec. 4.

1. Theoretical Analysis

1.1. Proofs of main theorems

In this section, we present the proof of theorems 4.2 and 4.3 from the main paper.

1.1.1. Notation and preliminaries

We fix a feedforward network of L layers and widths $\{n_0, n_1, \dots, n_L\}$ $f : \mathbb{R}^{n_0} \to \mathbb{R}^{n_L}$ defined by

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

where $T_i : x_i \to \theta_i x_i + b_i$ is an affine transformation with trainable weights $\theta_i \in \mathbb{R}^{n_i \times n_{i_i}}$, $b_i \in \mathbb{R}^{n_i}$, and ψ is a non-linear activation acting component wise. 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 this paper, we are primarily concerned with three main activations. Namely, Gaussian, sine and wavelet. A Gaussian is defined by $e^{-\frac{-\sigma^2 x^2}{2}}$ where $\frac{1}{\sigma^2}$ is called the variance of the Gaussian and a sine function by $sin(\omega x)$ where $\omega > 0$ is the frequency of the sine function. Finally, we will use the Gabor wavelet as in [9] which is denoted by $\psi(x; \omega_0; s_0)$, which is a complex valued function, with real part given by $cos(\omega_0 x)e^{-s_0^2 x^2}$ and imaginary part by $sin(\omega_0 x)e^{-s_0^2 x^2}$. All the signals we consider in this paper are real-valued. Therefore, we will be using the real part of the Gabor wavelet. As this will be the only wavelet we consider in this work we will simply call it a wavelet and drop the name Gabor.

For the following discussions we will assume our network only has weight parameters and no bias as this will make the mathematics more clear for the reader. However, we note that the proof of the main theorems go through for the case the network has biases with simple modifications.

We fix a data set, which we denote by $(X, Y) \in \mathbb{R}^{n_0 \times N} \times \mathbb{R}^{n_L \times N}$, consisting of input data $X \in \mathbb{R}^{n_0 \times N}$ and targets $Y \in \mathbb{R}^{n_L \times N}$. X will have 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.

An implicit neural representation can thus be viewed as a map

$$f : \mathbb{R}^{n_0 \times N} \times \mathbb{R}^p \to \mathbb{R}^{n_L \times N} \tag{2}$$

where p denotes the parameter dimension. In general, parameter vectors $\theta \in \mathbb{R}^p$ will be denoted in coordinates by as $\theta = (\theta(1), \dots, \theta(L))$, where each $\theta(i) \in \mathbb{R}^{n_i \times n_{i-1}}$ corresponds to the parameters of the ith-layer. Each such parameter $\theta(i)$

*equal contribution

represents a matrix of trainable weights given by:

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

Thus with the notation above parameters are represented as matrices. However, when we take derivatives with respect to parameter variables, we will to flatten the parameter 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

$$Vec(\theta(i)) = (\theta(i)_{11}, \ldots, \theta(i)_{1n_{i-1}}, \ldots, \theta(i)_{n_i1}, \ldots, \theta(i)_{n_in_{i-1}})^T. \tag{3}$$

We will always assume such flattening of matrices has been done and therefore we 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.

Feedforward neural networks are composed of their layers. We express this by writing

$$f = f_L \circ \cdots \circ f_1 \tag{4}$$

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

$$f_k : \mathbb{R}^{n_{k-1} \times N} \times \mathbb{R}^{n_k \times n_{k-1}} \times \cdots \times \mathbb{R}^{n_L \times n_{L-1}} \to \mathbb{R}^{n_k \times N} \times \mathbb{R}^{n_{k+1} \times n_k} \times \cdots \times \mathbb{R}^{n_L \times n_{L-1}} \tag{5}$$

is the map defined by

$$f_k(Z, \theta(k), \ldots, \theta(L)) = (\psi(\theta(k) \cdot Z), \theta(k+1), \ldots, \theta(L))^T. \tag{6}$$

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 (6), the map f_1 is a map

$$f_1 : \mathbb{R}^{n_0 \times N} \times \mathbb{R}^{n_1 \times n_0} \times \cdots \times \mathbb{R}^{n_L \times n_{L-1}} \to \mathbb{R}^{n_1 \times N} \times \mathbb{R}^{n_2 \times n_1} \times \cdots \times \mathbb{R}^{n_L \times n_{L-1}} \tag{7}$$

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

$$f_1 : \mathbb{R}^{n_1 \times n_0} \times \cdots \times \mathbb{R}^{n_L \times n_{L-1}} \to \mathbb{R}^{n_1 \times N} \times \mathbb{R}^{n_2 \times n_1} \times \cdots \times \mathbb{R}^{n_L \times n_{L-1}} \tag{8}$$

defined by

$$f_1(\theta(1), \ldots, \theta(L)) = (\psi(\theta(1) \cdot X), \theta(2), \ldots, \theta(L))^T \tag{9}$$

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.

For this section, we will fix our loss function as the MSE loss function. We write this loss function as

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} |f(x_i, \theta) - y_i|^2. \tag{10}$$

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:

$$c : \mathbb{R}^{n_L \times N} \to \mathbb{R} \tag{11}$$

defined by

$$c(z_1, \dots, z_N) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} |z_i - y_i|^2 \tag{12}$$

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 $c \circ f$. Theorems 4.2 and 4.3 also hold for more general loss functions such as Binary Cross Entropy (BCE) loss.

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

$$\mathcal{L} : \mathbb{R}^p \to \mathbb{R} \tag{13}$$

and therefore its differential is a map

$$D\mathcal{L} : \mathbb{R}^p \to Lin(\mathbb{R}^p, \mathbb{R}) \cong \mathbb{R}^{p \times 1} \tag{14}$$

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 (12).

Proposition 1.1. *$Dc(Z) = \frac{1}{N}(Z - Y)$, where Y denotes the matrix of targets from our data set.*

The next step is to compute the differential of the neural network function f. The differential will be a map

$$Df : \mathbb{R}^p \to Lin(\mathbb{R}^p, \mathbb{R}^{n_L N}) \cong \mathbb{R}^{p \times n_L N}. \tag{15}$$

By equation (4), and the chain rule, we have that for a vector $\theta \in \mathbb{R}^p$

$$Df(\theta) = Df_L(f_{L-1} \circ \cdots \circ f_1(\theta)) \cdot Df_{L-1}(f_{L-2} \circ \cdots \circ f_1(\theta)) \cdots Df_2(f_1(\theta)) \cdot Df_1(\theta). \tag{16}$$

We therefore need to compute the differential of each f_k.

We use the following notation to denote partial derivatives with respect to the space variable Z and the parameter variable $\theta(j)$, for $k \leq j \leq L$. 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 $k \leq j \leq L$. Thus for example, we have that

$$\frac{\partial}{\partial Z} \psi(\theta^j(k) \cdot Z) = \left(\frac{\partial}{\partial Z} \psi(\theta^j(k) \cdot Z_1), \dots, \frac{\partial}{\partial Z} \psi(\theta^j(k) \cdot Z_N) \right)^T. \tag{17}$$

The following lemma shows how variable derivatives of variable indices interact which each other.

Lemma 1.2. *$\frac{\partial}{\partial Z_j} \psi(\theta^i(k) Z_i) = 0$ 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 $j \neq i$. □

Proposition 1.3. *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 $1 \leq i \leq n_k$. 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 1.2 we have that $\frac{\partial}{\partial^j \theta(k)}\psi(\theta^i(k) \cdot Z) = 0$ for $i \neq j$. □

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

$$D^2\mathcal{L}(\theta) = Df(\theta)^T \cdot (\frac{1}{N} I) \cdot Df(\theta) + \frac{1}{N}(f(\theta) - Y) \cdot D^2 f \tag{18}$$

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

$$D\mathcal{L}(\theta) = Dc(f(\theta)) \cdot Df(\theta) \tag{19}$$

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

$$D^2\mathcal{L}(\theta) = Df(\theta)^T \cdot D^2c(f(\theta)) \cdot Df(\theta) + Dc(f(\theta)) \cdot D^2 f(\theta). \tag{20}$$

By proposition 1.1, 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. □

Proposition 1.4 implies that in order to compute the Hessian of the loss, we need to compute the Hessian of the neural network function. This can be done by the chain rule and induction.

Lemma 1.5. *We have the following decomposition for the Hessian of f*

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

Proof. The proof of this follows by induction on the layers. □

Lemma 1.5 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 1.3. We will now give a matrix formula for the Hessian of each layer f_k.

In order to compute the Hessian $D^2 f_k$, we will flatten Df_k so that it is a map

$$Df_k : \mathbb{R}^{n_{k-1} \times N} \times \mathbb{R}^{n_k \times n_{k-1}} \times \cdots \times \mathbb{R}^{n_L \times n_{L-1}} \to \mathbb{R}^{(n_k \times N + n_{k+1} \times n_k + \cdots + n_L \times n_{L-1})(n_{k-1} \times N + n_k \times n_{k-1} + \cdots + n_L \times n_{L-1})}. \tag{21}$$

Then for a point $(Z, \theta(k), \dots, \theta(L))$, we have that $D^2 f_k(Z, \theta(k), \dots, \theta(L))$ 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 1.3. We start by computing these square matrices.

Lemma 1.6. *Given the matrix representation of Df_k in proposition 1.3 and $1 \leq i \leq n_k N$, we have that the derivative of the ith-row of Df_k is given by*

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

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 1.3. 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 1.6, we can compute a full matrix representation of the Hessian of f_k.

Proposition 1.7. *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 1.6 and stacking them on top of each other. □

Proposition 1.3 shows that the gradient of the kth-layer neural function has some sparsity in the matrix representation. Furthermore, proposition 1.7 shows that the Hessian of the kth-layer neural function f_k has some sparsity, regardless of the activation. We will not consider such structures as sparse as they are shared by all implicit neural representations. Rather, we will show in the next section, that much more sparsity can be obtained in the Hessian of the loss when the neural network uses a ReLU activation, by showing that the derivative terms in 1.7 vanish.

1.1.2. Proofs of theorems 4.2 and 4.3 from the main paper

From propositions 1.4, 1.3, 1.7 and lemma 1.5 we see that in order to compute the Hessian of the MSE loss $\mathcal{L}$, we need to understand the first and second derivatives of our activation function.

Proposition 1.8.

1. *Let* $\phi(x) = sin(\omega x)$. *Then* $\phi'(x) = \omega cos(\omega x)$ *and* $\phi''(x) = -\omega^2 sin(\omega x)$.
2. *Let* $\phi(x) = e^{-\frac{\sigma^2 x^2}{2}}$. *Then* $\phi'(x) = -\sigma^2 x e^{-\frac{\sigma^2 x^2}{2}}$ *and* $\phi''(x) = -\sigma^2 e^{-\frac{\sigma^2 x^2}{2}} + \sigma^4 x^2 e^{-\frac{\sigma^2 x^2}{2}}$.

The proof of the above proposition is straightforward from standard differentiation rules along with the chain rule.

In the paper [9], a Gabor wavelet is used as an activation function for implicit neural representations. We remind the reader from our discussion in Sec. 1.1.1 that the wavelet we use in this work is the Gabor wavelet $\psi(x; \omega_0; s_0)$, which is a complex valued function, and its real part is given by $cos(\omega_0 x)e^{-s_0^2 x^2}$ and its imaginary part by $sin(\omega_0 x)e^{-s_0^2 x^2}$. Applying the chain rule and proposition 1.8 we get similar derivative formulae for the real and imaginary part of the Gabor wavelet. Therefore, for the following we will focus on sine, Gaussian and ReLU activations with the knowledge that the proofs for the sine and Gaussian case go through for the Gabor wavelet case by restricting to the real and imaginary parts of the wavelet. Since all our signals are real-valued the Gabor wavelet we use is the real part of the Gabor wavelet.

We will use some basic concepts from measure theory such as sets of measure zero. The reader who is not familiar with such material may consult the book [11].

Proposition 1.9. *Let* $\phi(x)$ *denote a sine, Gaussian or Gabor wavelet function. On any finite interval* I *in* $\mathbb{R}$, *the set of zeros of* ϕ, ϕ' *and* ϕ'' *have Lebesgue measure zero. In particular, normalizing the Lebesgue measure to have measure 1 on* I, *so that we obtain a probability measure, we have that the zeros of* ϕ, ϕ' *and* ϕ'' *have probability zero.*

Proof. The proof of this follows by first observing that on any finite interval ϕ only has finitely many zeros. Using proposition 1.8, we see that ϕ' and ϕ'' also have only finite zeros on a finite interval. The result follows. □

Proposition 1.10. *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 [8] 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. □

Proposition 1.11. *Let ϕ denote a ReLU activation. If I is an interval of finite non-zero Lebesgue measure that contains a sub-interval of negative numbers of non-negative measure. Then the zero set of ϕ and ϕ' has non-zero measure. If I is any interval of finite non-zero Lebesgue measure then the zero set of ϕ'' has non-zero Lebesgue measure. By normalizing the Lebesgue measure of I to be 1, we get that the probability of the zero set of ϕ and ϕ' is non-zero when I contains a sub-interval of negative numbers of non-zero probability. Furthermore, the probability of the zero set of ϕ'' is non-zero.*

Proof. The proof of this follows from proposition 1.10. □

Proof of theorem 4.2 from main paper. By propositions 1.8, 1.9 we get that the derivative terms in the formula 1.7, will be non-zero with probability 1. Then from propositions 1.4, 1.3, we have that the Hessian will in be a dense matrix. Therefore, given any non-zero vector v, we have that the Hessian vector product Hv will be a dense vector. □

Proof of theorem 4.3 from main paper. By propositions 1.10 and 1.11 we get that the the derivative terms in the formula 1.7, will be zero with high probability. Then from propositions 1.4, 1.3 we have that the Hessian will in be a sparse matrix. Therefore, given any non-zero vector v, we have that the Hessian vector product Hv will be a sparse vector. □

2. Algorithms

In section 4.1 of the main paper, we spoke about Kronecker factored preconditioners and mentioned that there were other algorithms that used other types of preconditioning techniques such as Gauss-Newton, L-BFGS and Shampoo. In this section we discuss the K-FAC algorithm that makes use of a Kronecker factored preconditioner and the Shampoo algorithm that makes use of left and right preoconditioning matrices.

2.1. A summary of K-FAC

Kronecker Factored Approximate Curvature (K-FAC) is a second order preconditoning algorithm introduced in [5]. The algorithm seeks to approximate the Hessian of an objective function via the Fisher information matrix [5]. In general, the Fisher information matrix is dense matrix, thus for implementation purposes this would require a huge memory cost for large parameter objective functions. Therefore, the authors impose that the approximation should factor into Kronecker products, where matrix forming the Kronecker product comes from the layer of the neural network.

For this section we fix an objective function f that we want to minimize. Let $G = J^T J$ denote the Gauss-Newton matrix of f, where $J = \nabla f^T$ is the Jacobian of f. When implementing K-FAC one must resort to the empirical Fisher information matrix, which is defined as as the expected value of G

$$F = \mathbb{E}(G) \approx \mathbb{E}(H). \tag{22}$$

Since G is a first order approximation of the hessian H of f, we see that the empirical Fisher information matrix approximates the expected value of the Hessian.

Given a neural network Φ 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

$$\mathbb{E}(\nabla_{\theta^i} f \otimes \nabla_{\theta^j} f) \tag{23}$$

where i and j are layers in the network and θ_i the parameters in layer i.

The idea of the algorithm is to now impose a Kronecker product structure on each such block and form the approximation

$$\mathbb{E}(\nabla_{\theta^i} f \otimes \nabla_{\theta^j} f) \approx \mathbb{E}(\widetilde{\phi}^{i-1}(\widetilde{\phi}^{j-1})^T) \otimes \mathbb{E}(\delta^i(\delta^j)^T) \tag{24}$$

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

We thus see that the empirical Fisher information matrix can be approximated by Kronecker products $\mathbb{E}(\widetilde{\phi}^{i-1}(\widetilde{\phi}^{j-1})^T) \otimes \mathbb{E}(\delta^i(\delta^j)^T)$. Abusing notation and denoting this approximation by F as well, the K-FAC algorithm update is

$$\theta_{k+1} = \theta_k - \eta F^{-1} \nabla_{\theta_i} f. \tag{25}$$

We thus see that K-FAC is a preconditioned gradient descent algorithm with preconditioner given by a Kronercker factored approximation to the empirical Fisher information matrix. For more details on the K-FAC algorithm we refer the reader to [5].

2.2. A summary of Shampoo

Shampoo is a preconditioning algorithm that optimizes objective functions over tensor spaces, introduced in [3]. In general, the algorithm works to optimize parameters that can represented as general tensors. In this regard, the optimizer is extremely effective when working on general neural architectures such as convnets. Implicit neural representations (INRs) are represented by feed forward neural networks, and hence their parameters are represented by a 2-tensor i.e. a matrix. Therefore, we will be analyzing Shampoo in the setting that our parameters are represented as matrices. However, we point out to the reader that the algorithm does work on more general tensors, see [3] for details.

As explained in section 3.1, a preconditioner is in general a matrix that acts on the gradient before an update is performed. This action is done by matrix multiplication on the left of the gradient. Let us denote the preconditioner by L, then the action if given by

$$L \cdot g \tag{26}$$

We can also act on the gradient on the right. Given a matrix R, we could also do

$$g \cdot R. \tag{27}$$

Combining (26) and (27) we obtain the left and right preconditioning transformation

$$L \cdot g \cdot A. \tag{28}$$

If we fix an objective function $f : \mathbb{R}^{m\times n} \to \mathbb{R}$, e.g. the MSE loss of a neural network. The shampoo algorithm for f (in the case of 2-tensors) takes as L_t the Gauss-Newton matrix $J^T J$, an $m \times m$-matrix, and as R_t the matrix JJ^T, an $n \times n$-matrix. The pseudocode for Shampoo (in the case of 2-tensors) is given in algorithm 1.

Algorithm 1 Shampoo

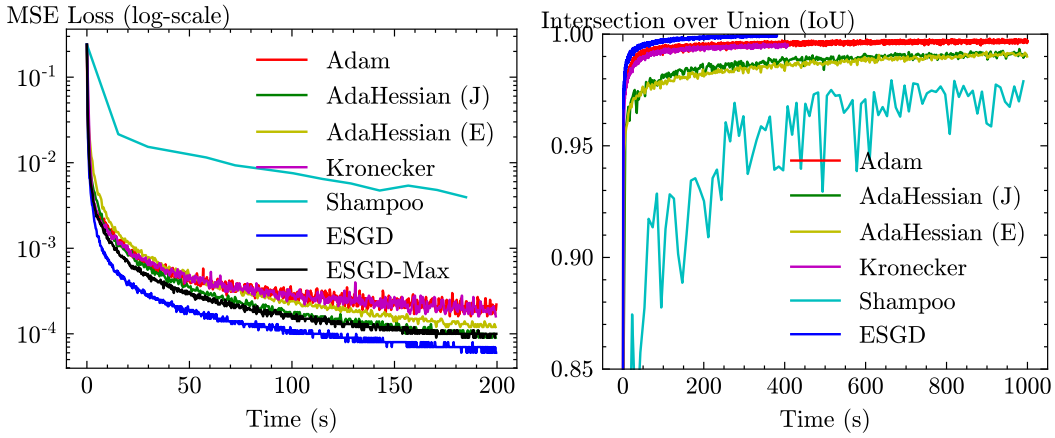
Require: initial point x_0, iterations N, learning rate η, $\epsilon > 0$.
  $L_0 \leftarrow \epsilon I_m$
  $R_0 \leftarrow \epsilon I_n$
  for $t = 0, \cdots, N$ **do**
    $g_t \leftarrow \nabla_{x_t} f$
    $L_t \leftarrow L_{t-1} + J_t^T J_t$
    $R_t \leftarrow R_{t-1} + J_t J_t^T$
    $x_{t+1} \leftarrow x_t - \eta L_t^{-1/4} g_t R_t^{-1/4}$
  end for

For the experiments carried out in this paper (including supplementary material) we will make use of the Shampoo algorithm for 2-tensors, as INRs are feedforward networks. For details on Shampoo for general tensors we refer the reader to the paper [3].

3. Experiments

In Section 5 of the main paper, we focused our analysis on Gaussian neural fields. In this section we extend our analysis to other activations such as sine, wavelet and ReLU. We also present results on additional datasets.

3.1. Additional results: Time-based comparisons



(a) 2D image reconstruction

(b) 3D shape reconstruction

Figure 1. Comparison of training convergence (**in time**) of a *Gaussian-activated neural field* for various preconditioners on 2D image reconstruction and (Fig. 1a) and 3D shape reconstruction task (Fig. 1b).

3.2. Additional results: 2D Image Reconstruction

3.2.1. Quantitative results: Comparison of training convergence for various preconditioners on sine, wavelet, ReLU-PE and ReLU networks.

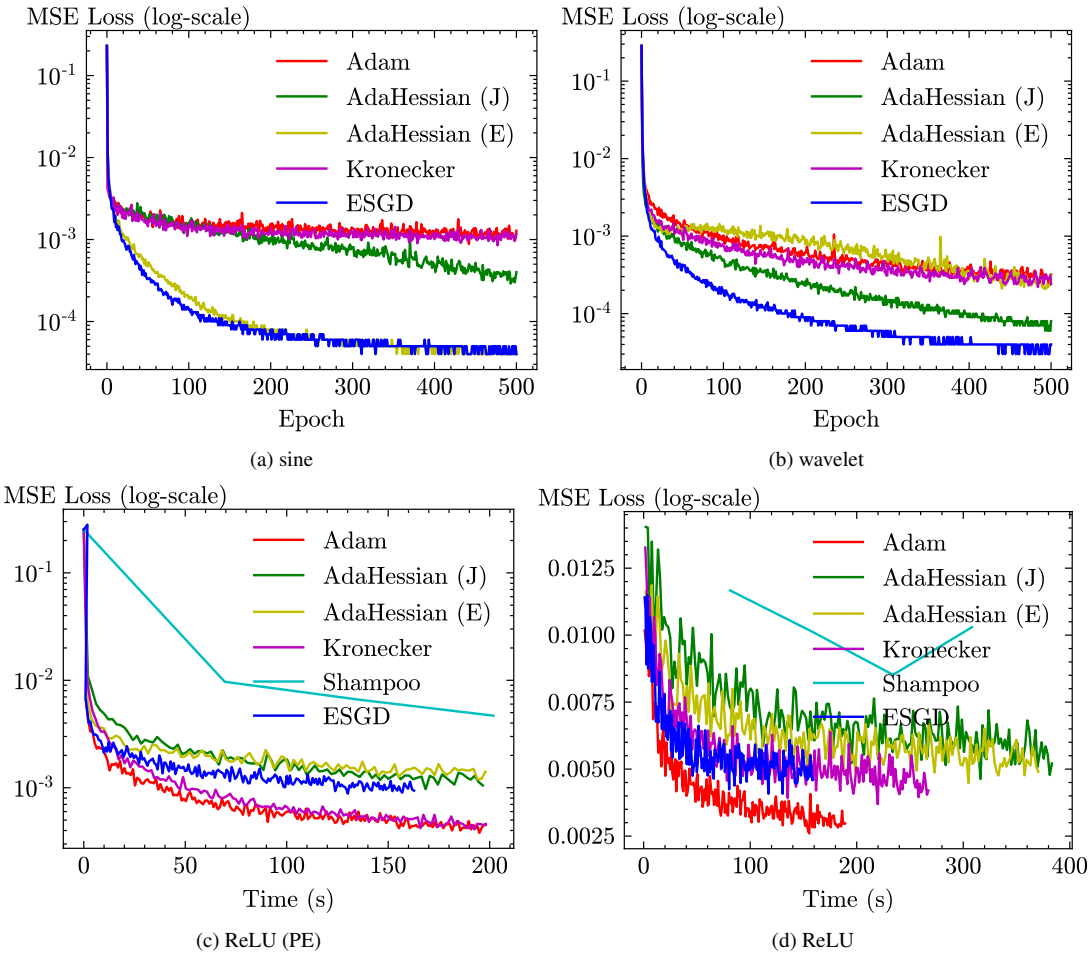
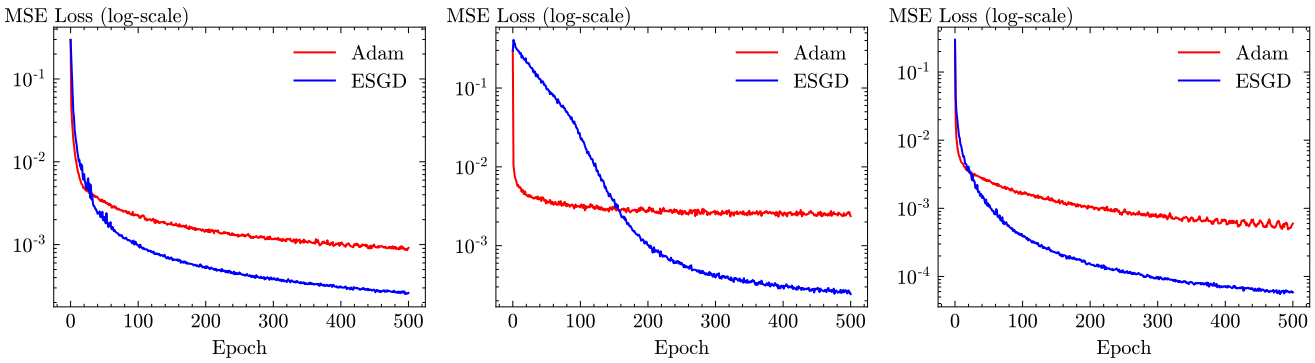


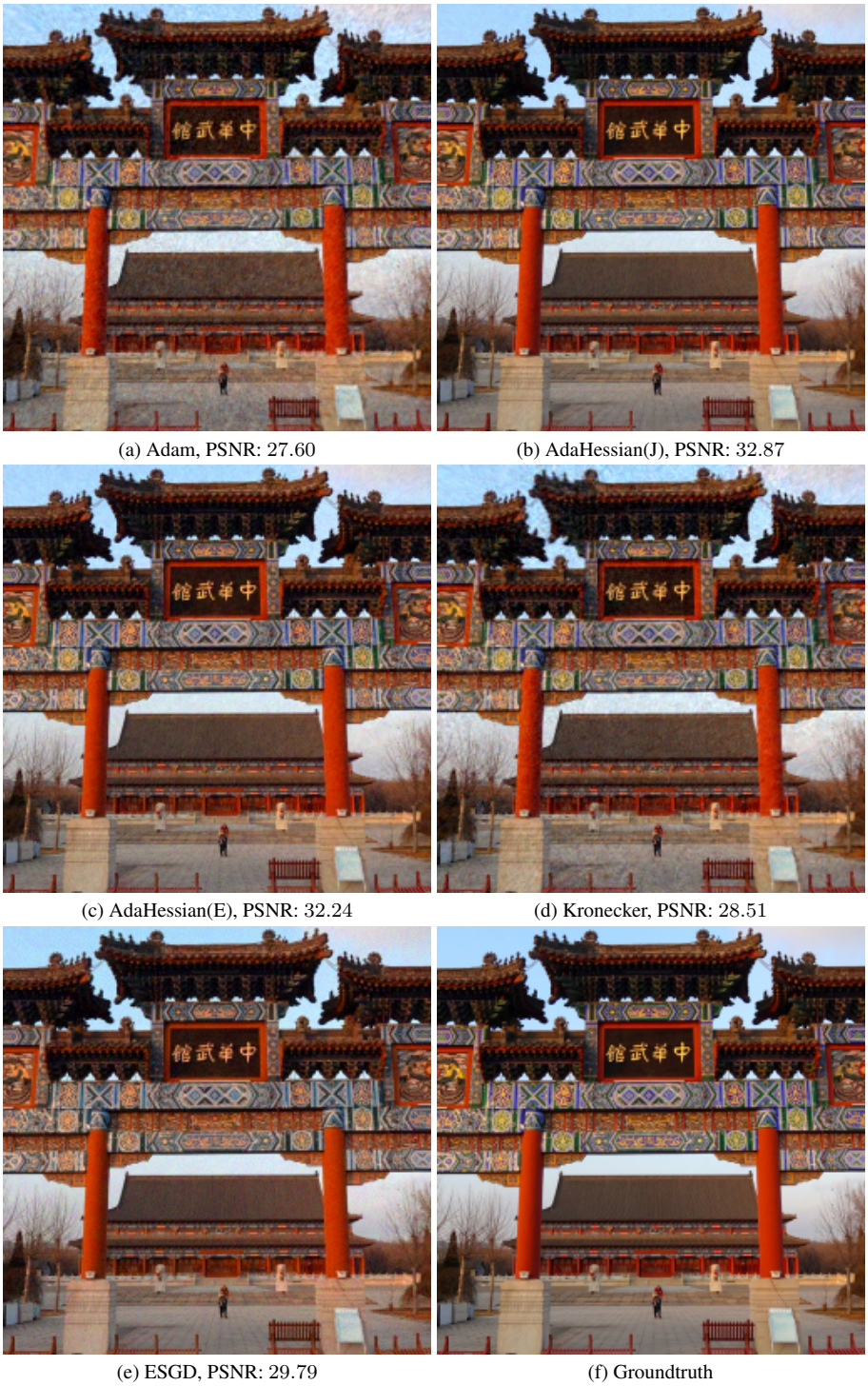
Figure 2. **Comparison of training convergence for various preconditioners on sine and wavelet-activated neural fields.** We evaluated each neural field on *lion* instance from the DIV2K dataset. ESGD consistently demonstrates superior convergence compared to other preconditioners on other activations. **Note:** The analysis for Gaussian network is available in Fig. 5 in the main paper.



(a) Gaussian (b) sine (c) wavelet

Figure 3. **Comparison of training convergence for ESGD vs. Adam optimizer on Gaussian, sine and wavelet-activated neural fields.** We evaluated each neural field on **all** 14 test instances from the DIV2K dataset. Interestingly, we observed that sine-based network trained with Adam reached a lower loss than those trained with ESGD during the beginning of the optimization. However, ESGD eventually caught up and converged to a lower minima compared to Adam. Overall, ESGD demonstrates superior convergence compared to Adam on all activations. **Note**: The plot shows the mean MSE loss averaged across 14 runs.

3.2.2. Qualitative results: Gaussian



(a) Adam, PSNR: 27.60 (b) AdaHessian(J), PSNR: 32.87

(c) AdaHessian(E), PSNR: 32.24 (d) Kronecker, PSNR: 28.51

(e) ESGD, PSNR: 29.79 (f) Groundtruth

Figure 4. Comparison of the intermediate reconstructions produced by Gaussian-activated neural field at epoch 300.

3.2.3. Qualitative results: sine

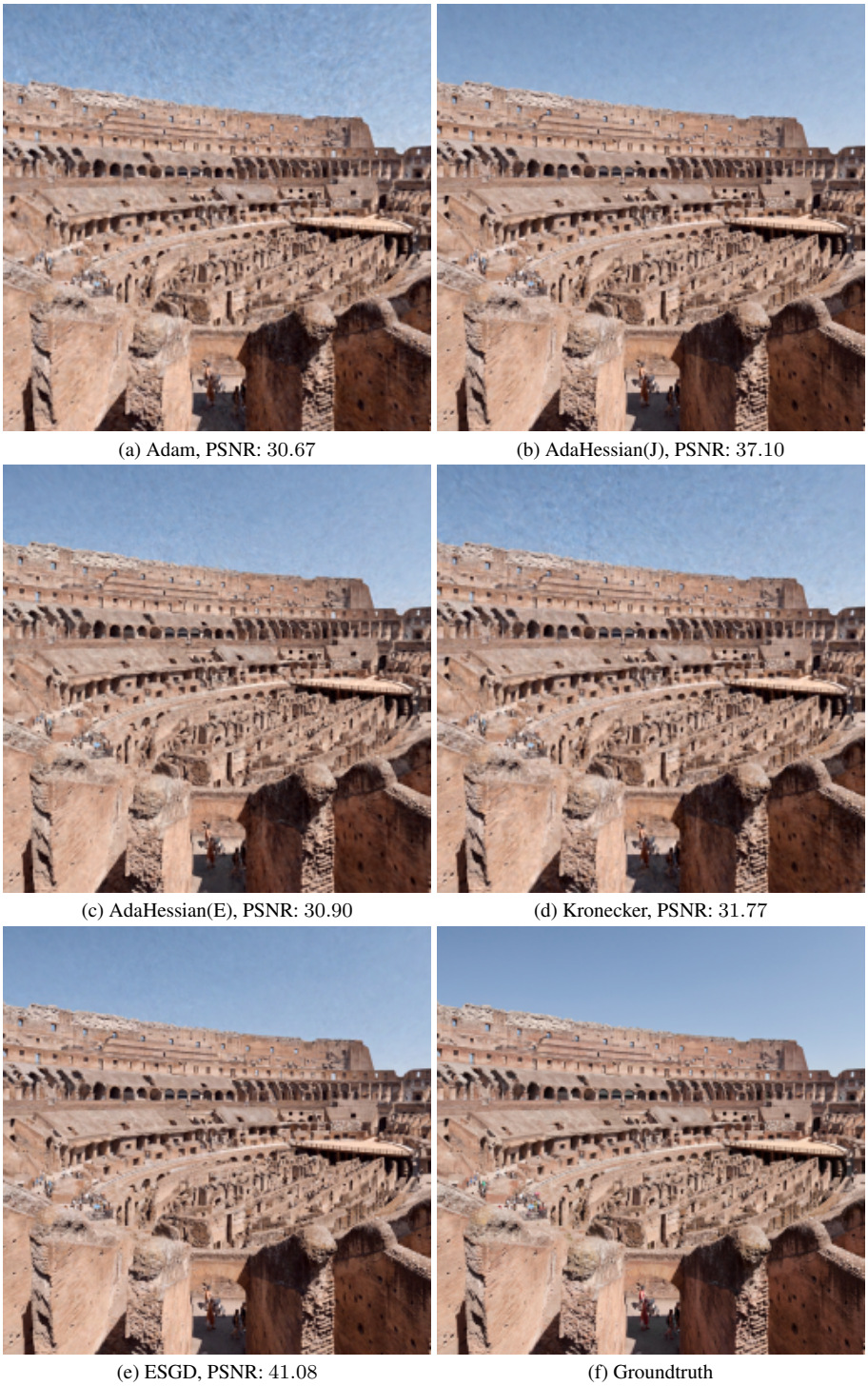
(a) Adam, PSNR: 26.11 (b) AdaHessian(J), PSNR: 25.92

(c) AdaHessian(E), PSNR: 34.71 (d) Kronecker, PSNR: 26.59

(e) ESGD, PSNR: 36.16 (f) Groundtruth

Figure 5. Comparison of the intermediate reconstructions produced by sine-activated neural field at epoch 300.

3.2.4. Qualitative results: wavelet



(a) Adam, PSNR: 30.67 (b) AdaHessian(J), PSNR: 37.10

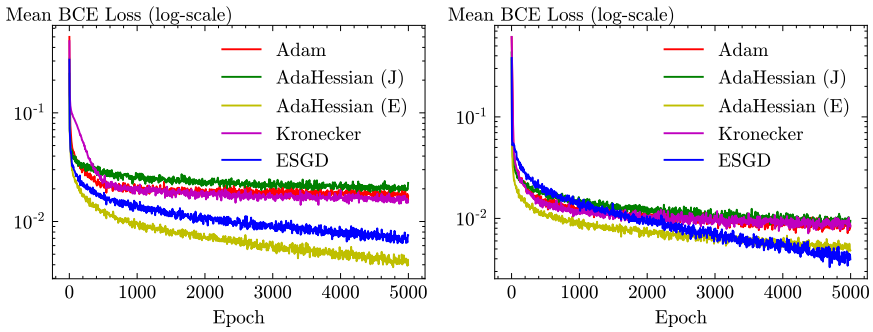
(c) AdaHessian(E), PSNR: 30.90 (d) Kronecker, PSNR: 31.77

(e) ESGD, PSNR: 41.08 (f) Groundtruth

Figure 6. Comparison of the intermediate reconstructions produced by wavelet-activated neural field at epoch 300.

3.3. Additional results: 3D Shape Reconstruction

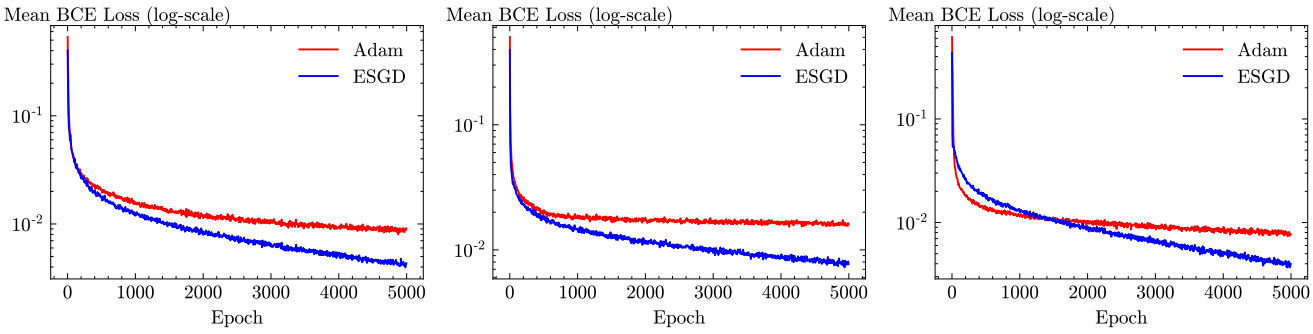
3.3.1. Quantitative results: Comparison of training convergence for various preconditioners



(a) sine (b) wavelet

Figure 7. **Comparison of training convergence for various preconditioners on sine and wavelet-activated neural fields.** We evaluated each neural field on *armadillo* test instances from the Stanford dataset. ESGD demonstrates superior convergence compared to other preconditioners, striking a balance between accuracy and computational efficiency. **Note:** The analysis for Gaussian network is available in Fig. 6 in the main paper.

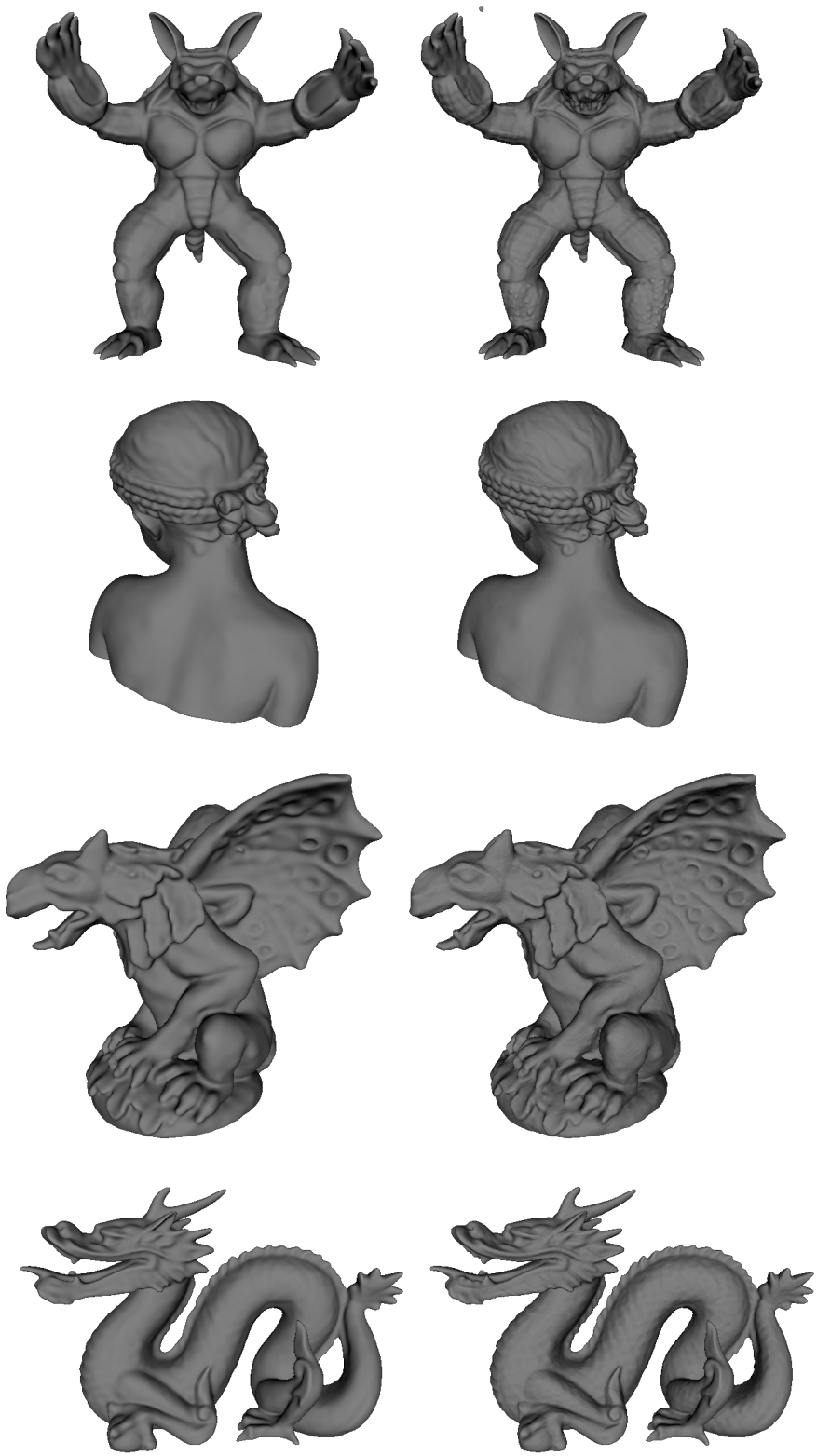
3.3.2. Quantitative results: Comparison of training convergence for Adam and ESGD



(a) Gaussian (b) sine (c) wavelet

Figure 8. **Comparison of training convergence for ESGD vs. Adam on Gaussian, sine and wavelet-activated neural fields.** We evaluated each neural field on **five** instances (*armadillo, bunny, bimba, dragon, gargoyle*) from the Stanford dataset. Overall, ESGD demonstrates superior convergence compared to Adam. **Note**: The plot shows the mean BCE loss averaged across 5 runs.

3.4. Qualitative Results



(a) Adam (b) ESGD

Figure 9. Compared to Gaussian-activated network trained with Adam (Fig. 9a), ESGD (Fig. 9b) has reconstructed the shapes with significantly improved fidelities at epoch 500. (*zoom in* $4\times$ for better clarity.)

3.5. Additional results: Neural Radiance Fields (NeRF)

3.5.1. Results on Blender dataset

Scene	Train PSNR	**Adam** Iteration ↓	Time (s)↓	Test PSNR ↑	Test SSIM ↑	Test LPIPS ↓	**ESGD-Max** Iteration ↓	Time (s)↓	Test PSNR ↑	Test SSIM ↑	Test LPIPS ↓
lego	26.50	200K	341.81	24.93	0.86	0.12	140K	**191.00**	25.37	0.87	0.11
chair	29.95	200K	339.29	32.79	0.97	0.04	140K	**182.71**	33.37	0.98	0.03
drums	23.46	200K	356.33	23.73	0.86	0.16	150K	**195.21**	24.38	0.88	0.13
hotdog	33.86	200K	352.58	30.69	0.94	0.08	100K	**132.40**	31.28	0.95	0.07
ficus	26.00	200K	346.23	25.10	0.91	0.08	140K	**181.14**	25.93	0.92	0.07
materials	28.62	120K	**115.34**	24.65	0.89	0.09	200K	260.78	24.30	0.88	0.11
ship	28.58	140K	**134.47**	28.06	0.78	0.18	200K	244.31	27.17	0.74	0.25
mic	32.97	170K	**164.00**	33.22	0.97	0.03	200K	166.68	32.94	0.97	0.03

Table 1. Quantitative results of **Gaussian-NeRF** on instances from the 360° **BLENDER** dataset [7]. Note that the report time solely refers to the optimization update step. Training convergence is determined based on the training curve displayed on TensorBoard, with a smoothing factor of $0.8 - 0.9$ applied for better visualization.

3.5.2. Results on Bleff dataset

Scene	Train PSNR	Adam Iteration ↓	Adam Time (s)↓	Adam Test PSNR ↑	Adam Test SSIM ↑	Adam Test LPIPS ↓	ESGD-Max Iteration ↓	ESGD-Max Time (s)↓	ESGD-Max Test PSNR ↑	ESGD-Max Test SSIM ↑	ESGD-Max Test LPIPS ↓
balls	37.78	200K	522.56	37.11	0.93	0.08	150K	**392.65**	37.15	0.93	0.09
deer	47.50	150K	**359.30**	46.73	0.99	0.02	150K	360.76	46.53	0.99	0.02
chair	36.38	180K	492.08	36.27	0.90	0.17	180K	**485.32**	36.02	0.89	0.18
root	38.58	200K	501.13	37.4	0.98	0.03	150K	**394.45**	37.38	0.98	0.03
roundtable	48.25	160K	367.38	47.36	1.00	0.01	150K	**341.11**	46.90	0.99	0.01

Table 2. Quantitative results of **Gaussian-NeRF** on instances from the **BLEFF** dataset [12]. Note that the report time solely refers to the optimization update step. Training convergence is determined based on the training curve displayed on TensorBoard, with a smoothing factor of $0.8 - 0.9$ applied for better visualization.

3.5.3. Novel View Synthesis Results for Sec. 5.3 in the main paper

(a) 1^{st} View

(b) 50^{th} View

(c) 1^{st} View

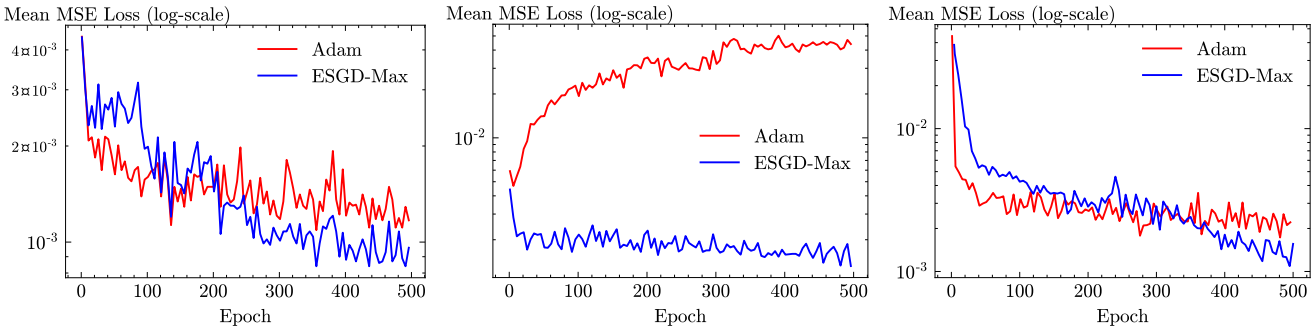
(d) 50^{th} View

(e) 1^{st} View

(f) 50^{th} View

Figure 10. Novel view synthesis results using Gaussian network trained with ESGD-Max.

3.6. Additional results: Video Reconstruction



(a) Gaussian (b) sine (c) wavelet

Figure 11. **Comparison of training convergence for ESGD vs. Adam on Gaussian, sine and wavelet-activated neural fields.** We evaluated each neural field on the *bikes* sequence available from the scikit-video. Interestingly, we observed that Gaussian and wavelet networks trained with Adam reached a lower loss than those trained with ESGD during the beginning of the optimization. However, ESGD eventually caught up and converged to a lower minima. In contrast, sine-activated networks trained with Adam diverged rapidly after several epochs. We speculated that this divergence may be due to the initialization method, which could cause the network weights to become poorly conditioned in video reconstruction task. ESGD did not exhibit the same divergence, likely benefiting from better preconditioning when computing the updates.

4. Reproducibility and Implementation Details

4.1. 2D Image Reconstruction

Data. We evaluated on **all** the test instances from DIV2K dataset [1]. We resized the original images, initially captured at a resolution of 512×512 pixels, to a resolution of 256×256 pixels. We used minibatch size of 512 during training.

Architecture. We used a 5-layer network, where each hidden layer contains 256 neurons. For Gaussian-activated neural field, we used Gaussian activation defined as $\phi(\mathbf{x}) = \exp(\frac{-0.5\mathbf{x}^2}{\sigma^2})$, and we used $\sigma = 0.05$. For sine-activated neural field, we used sine activation defined as $\phi(\mathbf{x}) = \sin(\omega x)$, and we used $\omega = 30$. As for wavelet-activated neural field, we employed wavelet activation defined as $\phi(\mathbf{x}) = \mathbf{x} \rightarrow \cos(\omega_0 \mathbf{x}) \exp(-s_0^2 \mathbf{x}^2)$. We used $\omega_0 = 10$ and $s_0 = 10$. We used $\phi(\mathbf{x}) = \{[\cos(2^k \pi \mathbf{x}), \sin(2^k \pi \mathbf{x})]\}_{k=0}^{L-1}$ for the positional encoding, where $L = 8$.

Initialization. We used default PyTorch initialisation (Kaiming Uniform) for Gaussian and wavelet-activated neural fields. As for sine-activated neural field, we employed the initialization proposed by Sitzmann et al. [10].

Hyperparameters. Unless specified, we used the default parameter values for each optimizer. For the Adam optimizer, we used a learning rate of 1×10^{-4} for Gaussian, sine and wavelet-activated neural fields and a learning rate of 1×10^{-3} for ReLU-PE network. For AdaHessian(J) and AdaHessian(E), we set the learning rate to 0.15 and used a hessian power of 1. For Kronecker-based preconditioner, we used a learning rate of 0.1 and performed the inverse update every 100 iterations. For Shampoo, we used a learning rate of 0.1 and 0.01 for Gaussian/wavelet networks and sine network, respectively. As for ESGD, we chose a learning rate of 0.15. We incorporated a warmup strategy. Specifically, we implemented the equilbrated gradient preconditioning for the first 50 iterations as a warmup phase, followed by subsequent updates every N iterations, with N set to 100.

Hardware. We ran all experiments on a NVIDIA 4090 GPU with 24Gb of memory.

4.2. 3D Shape Reconstruction

Data. We use the *Armadillo*, *Bunny*, *Bimba*, *Gargoyle* and *Dragon* from the Stanford 3D Scanning Repository [1]. For training, we sampled one million 3D points – one-third of the points were sampled uniformly within the volume, and remaining two-thirds of the points were sampled near the mesh surface and perturbed with random Gaussian noise using sigma of 0.1 and 0.01, respectively.

Architecture. We used a 5-layer network, where each hidden layer contains 256 neurons. For Gaussian-activated neural field, we used $\sigma = 0.09$. For sine network, we used $\omega = 30$. As for wavelet network, we used $\omega_0 = 10$ and $s_0 = 10$. For the ReLU-PE network, we used $\phi(\mathbf{x}) = \{[\cos(2^k \pi \mathbf{x}), \sin(2^k \pi \mathbf{x})]\}_{k=0}^{L-1}$ for the positional encoding, where $L = 8$.

Initialization. We used default PyTorch initialisation (Kaiming Uniform) for Gaussian-INR and ReLU/ReLU-PE INRs in all the experiments. As for sine-activated neural field, we employed the initialization proposed by Sitzmann et al. [10].

Hyperparameters. Unless specified, we used the default parameter values for each optimizer. For the Adam optimizer, we used a learning rate of 1×10^{-4}. For AdaHessian(J) and AdaHessian(E), we set the learning rate to 0.15 and used a hessian power of 1. For Kronecker-based preconditioner, we used a learning rate of 0.1, Tikhonov regularization parameter of 0.01, and performed the inverse update every 100 iterations. For Shampoo, we used a learning rate of 0.1. As for ESGD, we chose a learning rate of 0.15. We incorporated a warmup strategy. Specifically, we implemented the equilbrated gradient preconditioning for the first 50 iterations as a warmup phase, followed by subsequent updates every N iterations, with N set to 100.

Hardware. We ran all experiments on a NVIDA 4090 GPU with 24Gb of memory.

4.3. Neural Radiance Fields (NeRF)

In this section, we provide the reproducibility and implementation details for the experiments in **Sec. 5.3 of the main paper**, as well as those in Sec. 3.5.1 for the Blender 360 [7] dataset as well and Sec. 3.5.2 for the Blender Forward-Facing Dataset (BLEFF).

4.3.1. Forward-Facing dataset LLFF dataset [6]

Training details. We resized the images to 480×640 pixels. We trained all models for 200K iterations. As in [2, 4], we trained a single model without additional hierarchical sampling. During each optimization step, we randomly sampled 2048 pixel rays, and integrated 128 points along each ray.

Hyperparameters. Unless specified, we used the default parameter values for each optimizer. For the Adam optimizer, we used a learning rate of 1×10^{-4} and customised step learning rate schedule. For the ESGD optimizer, we started with a learning rate of 1 and exponentially decayed it to 0.01.

Hardware. The experiments are ran on a6000 GPU.

4.3.2. Blender $360°$ dataset [6]

Training details. We resized the images to 400×400 pixels. We trained all models for 200K iterations. Following the approach in [2, 4], we trained a single model without additional hierarchical sampling. For each optimization step, we randomly sampled 2048 pixel rays, and integrated 128 points along each ray.

[1] http://graphics.stanford.edu/data/3Dscanrep

Hyperparameters. Unless specified, we used the default parameter values for each optimizer. For the Adam optimizer,we started with a learning rate of 1×10^{-4} and exponentially decayed it to 1×10^{-6}. For the ESGD optimizer, we started with a learning rate of 1 and exponentially decayed it to 0.1.

Hardware. The experiments are ran on a a6000 GPU.

4.3.3. Blender Forward-Facing Dataset (BLEFF) dataset [12]

Training details. We resized the images to 400×400 pixels. We trained all models for 200K iterations. Following the approach in [2, 4], we trained a single model without additional hierarchical sampling. For each optimization step, we randomly sampled 2048 pixel rays, and integrated 128 points along each ray.

Hyperparameters. Unless specified, we used the default parameter values for each optimizer. For the Adam optimizer,we started with a learning rate of 1×10^{-4} and exponentially decayed it to 1×10^{-6}. For the ESGD optimizer, we started with a learning rate of 1 and exponentially decayed it to 0.01.

Hardware. The experiments are ran on a a6000 GPU.

References

[1] Eirikur Agustsson and Radu Timofte. Ntire 2017 challenge on single image super-resolution: Dataset and study. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2017. 21

[2] Shin-Fang Chng, Sameera Ramasinghe, Jamie Sherrah, and Simon Lucey. Gaussian activated neural radiance fields for high fidelity reconstruction and pose estimation. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXXIII*, pages 264–280. Springer, 2022. 22, 23

[3] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 1842–1850. PMLR, 2018. 8

[4] Chen-Hsuan Lin, Wei-Chiu Ma, Antonio Torralba, and Simon Lucey. Barf: Bundle-adjusting neural radiance fields. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5741–5751, 2021. 22, 23

[5] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015. 7, 8

[6] Ben Mildenhall, Pratul P Srinivasan, Rodrigo Ortiz-Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *ACM Transactions on Graphics (TOG)*, 38(4):1–14, 2019. 22

[7] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021. 18, 22

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

[9] Vishwanath Saragadam, Daniel LeJeune, Jasper Tan, Guha Balakrishnan, Ashok Veeraraghavan, and Richard G Baraniuk. Wire: Wavelet implicit neural representations. *arXiv preprint arXiv:2301.05187*, 2023. 1, 6

[10] V. Sitzmann, J. Martel, A. Bergman, D. Lindell, G., and Wetzstein. Implicit Neural Representations with Periodic Activation Functions. In *NIPS*, 2020. 21, 22

[11] Elias M Stein and Rami Shakarchi. Measure theory, integration, and hilbert spaces, 2005. 6

[12] Zirui Wang, Shangzhe Wu, Weidi Xie, Min Chen, and Victor Adrian Prisacariu. Nerf–: Neural radiance fields without known camera parameters. *arXiv preprint arXiv:2102.07064*, 2021. 19, 23