Neural-based General Matrix Inversion: Why can't, and Where can

Anonymous Authors[1]

Abstract

Deep neural networks have succeeded in scientific computing, especially in accuracy on specific tasks. Computing matrix inversion accurately and efficiently is an important problem in scientific computing and has many applications. However, until now, there is *no* neural-based general matrix inversion method. In this paper, we give a theoretical analysis of why modern neural networks can *not* form a general matrix inversion model. To state our analysis, we give a generalization of the Lipschitz function class to encompass a wider range of neural-based models. This paper also shows in which area, matrix inversion can be computed accurately by the trained neural network. To verify the correctness of our theoretical work, we conduct a series of experiments on different matrix datasets and analyze what the trained neural networks learn in the matrix inversion problem.

1. Introduction

In recent years, neural network-based methods have made advancements in many problems in scientific computing. Deep neural networks have been applied to eigenvalue problems for linear and semilinear second-order differential operators in high dimensions (Han et al., 2020). Also, for the eigenvalue problem of differential self-adjoint operators, a neural network–based solver approach was introduced in (Ben-Shaul et al., 2023). The most noteworthy result is that recent researchers used reinforcement learning with neural networks finding several fast matrix multiplication algorithms(Fawzi et al., 2022).

There is an important problem in scientific computing and computational mathematics: how to compute matrix inversion accurately and efficiently. In the past eras, researchers have invested a lot of effort in this problem. Methods such as LU decomposition, Cholesky decomposition, QR decomposition, and the Gauss-Jordan method provide efficient and stable strategies for matrix inversion.

The general matrix inversion model refers to a model that can approximate the matrix inversion operation with a small error in a large scale area of the whole space $\mathbb{R}^{n\times n}$. Although some recent research claims that neural networks can be used on matrix inversion tasks accurately, they are so far away from an end-to-end neural-based general matrix inversion model. (Jang et al., 1987; Fa-Long & Zheng, 1992; Steriti et al., 1990) all stated that they provide a neural-based method for matrix inversion, but their methods can only hold on an assigned training area. When the data is out of the area or their models encounter a specific matrix, the performance will be collapsed. (Fa-Long & Zheng, 2009) tried to use a neural network with newton iteration, but this method still only works for a specific area. (Li & Hu, 2022; Almasadeh et al., 2022) also needs to be combined with optimization techniques, which may not converge on specific data. Some recent papers (Dai et al., 2023; Gerontitis et al., 2023; Dai et al., 2022) show neural-based models perform well on time-varying matrix inversion problems, but this problem is completely different from the matrix inversion because the problem contains time as a variable and ground truth solution of the initial time is given.

As a result, we try to find whether there is a neural network model for the general matrix inversion problem and we debunk the existence of such a model in this paper. Previous research (Kim et al., 2021; Virmaux & Scaman, 2018; Latorre et al., 2020) relied on Lipschitz continuity to interpret the ability of the neural network. But, several modern neural networks are not Lipschitzable. To demonstrate our proofs, we need to find a much more universal property of modern neural networks. As a result, we give a generalization of the Lipschitz function class in this paper and use such properties to complete our proofs.

To support our proofs better, we select several specific datasets and train end-to-end neural networks on them. After finishing training, we analyze the trained model theoretically to find what the end-to-end models learned and how will they perform in the space of selected datasets.

In this paper, our main contributions are:

- We give a generalized Lipschitz continuity class, which

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

can contain most neural-based models mathematically while the traditional Lipschitz continuity can not.

- We first prove why there are no neural-based end-to-end matrix inversion models that achieve acceptable accuracy in $R^{n\times n}$ under mild assumptions.
- We prove in which specific area, neural networks can approximate the matrix inversion perfectly. Also, we analyze what the neural networks learn in the matrix inversion problem experimentally and theoretically.

2. Lipschitz Continuity and a Generalization

Mathematically, we define the function of matrix inversion as $Inv(x)$, where x is the input matrix. Also, we define the trained neural network model as $F(x)$, and the input of the neural network is the matrix and the output is the matrix inversion.

We represent the L_1 norm, L_2 norm, and L_∞ norm metrics by $||\cdot||_L$, because in our proof, only shareable properties are needed.

Suppose the dimension of the input matrix is fixed as $n \times n$. Obviously, the set of $n \times n$ singular matrices is 0-measure in $\mathbb{R}^{n\times n}$ under Lebesgue measurement. Hence, without loss of generality, we assume that the given dataset M does contain any singular matrices. Also, it is needed that dataset M must be positive-valued Lebesgue measurement in $\mathbb{R}^{n\times n}$ and sufficiently large.

We define the term 'fit' as that, the trained-well neural network model $F(x)$ can approximate the target function $Inv(x)$ under the formula:

$$\mathbb{E}_M(||Inv(x) - F(x)||_L^K) < E, \tag{1}$$

where $\mathbb{E}_M$ represents the expectation over dataset M, and K, E are constants.

For example, if we choose the L_2 norm and $K = 2$, the Equation 1 represents that, the mean-square-error between the output of trained neural network model and the target should be smaller than E.

The other kind of evaluation formula like $\mathbb{E}_M(\frac{||Inv(x)-F(x)||_L^K}{||x||_L^{K'}})$ will be discussed in Appendix A. People could use our proof techniques to consider other evaluation formulas in the other specific tasks.

Then, we need to introduce the Lipshitz continuity as

Definition 2.1. Given two metric spaces (X, dX) and (Y, dY), a function $f : X \to Y$ is called Lipschitz continuous (or K-Lipschitz) if there exists a constant $K \geq 0$ such that

$$d_Y(f(x_1), f(x_2)) \leq K d_X(x_1, x_2) \forall x_1, x_2 \in X \tag{2}$$

However, many neural network structures are not Lipschitz functions. Previous research (Kim et al., 2021) showed that multi-head dot-product attention is not Lipschitzable. Also, polynomials with degree ≥ 2 are not Lipshitz functions. Hence, we need to generalize the Lipshitz continuity.

In order to introduce our proof clearly, we generalize the Lipschitz continuity to polynomial Lipschitz continuity under the space $\mathbb{R}^n$ and metric L_p.

Definition 2.2. A function $f(x) : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}$ is called polynomial Lipschitz continuous function under two norm L^1, L^2 on $\mathbb{R}^{n_1}, \mathbb{R}^{n_2}$ if it satisfies

$$||f(x)-f(y)||_{L^2} \leq \sum_{i=0}^{n^f} f_i(||x||_{L^1}, ||y||_{L^1})||x-y||_{L^1}^i, \tag{3}$$

for any $x, y \in \mathbb{R}^n$, where $f_i(||x||_{L^1}, ||y||_{L^1})$ is a polynomial with variable $||x||_{L^1}, ||y||_{L^1}$ and n^f is constant, depend on function $f(x)$.

This definition is also a generalization of Hölder continuity.

3. Main Theorem: Why can't

In this section, we briefly summarize the proof of the main theorem in our paper. The full proof can be found in Appendix A.

The main theorem can be stated as follows:

Theorem 3.1. *Suppose the data is sampled from a dataset M in $\mathbb{R}^{n\times n}$ under mild constraints. Then, under either L_1 norm, L_2 norm, or L_∞ norm metric, any polynomial Lipschitz continuous neural network-based model can NOT fit the matrix inversion problem well over the dataset.*

Mathematically, we can state the theorem as

Theorem 3.2. *Suppose the data is sampled from a dataset M in $\mathbb{R}^{n\times n}$, with no singular matrices contained, and $B(\vec{a}, c)$ is contained in the set, which is a ball area and $\vec{a}$ in a data point, c is a sufficiently large number. Then, under either L_1 norm, L_2 norm, or L_∞ norm metric, any trained polynomial Lipschitz continuous neural network-based model $F(x)$ can NOT satisfy $\mathbb{E}_M(||Inv(x) - F(x)||_L^K) < E$ under Lebesgue measurement if $K > n^2$ for any E.*

To prove Theorem 3.2, we introduce some needed lemma first.

First, we want to introduce that the composition of two polynomial Lipschitz continuous functions is also polynomial Lipschitz continuous.

Lemma 3.3. *Suppose we have two functions $f(x) : \mathbb{R}^{n_2} \to \mathbb{R}^{n_3}$, $g(x) : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}$. And, Under norm L, either L_1, L_2 or L_∞, defined on $\mathbb{R}^{n_i}$, f, g are all polynomial Lipschitz continuous functions. Then the composition $f \circ g$ is also a polynomial Lipschitz continuous function.*

The proof can be found in Appendix A.1.

We also want to show that the combination of two polynomial Lipschitz continuous functions is also polynomial Lipschitz continuous.

Lemma 3.4. *Suppose we have two functions $f(x) : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}$, $g(x) : \mathbb{R}^{n_1} \to \mathbb{R}^{n_3}$. And, Under norm L, either L_1, L_2 or L_∞, defined on $\mathbb{R}^{n_i}$, f, g are all polynomial Lipschitz continuous functions. Then the combination $(f, g)(x) : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2+n_3}$ is also a polynomial Lipschitz continuous function.*

The proof can be found in Appendix A.2.

Before we prove some modern neural network structures are polynomial Lipschitz continuous, we need a lemma to reveal the connection between the elements in the Jacobian of the functions and the polynomial Lipschitz continuous property.

Lemma 3.5. *Suppose function $f(x) : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}$ and Jacobian of f exists everywhere. If the value of each element of the Jacobian is bounded by a polynomial of $||x||_L$, f is a polynomial Lipschitz continuous function.*

Full proofs are in Appendix A.3. We show the proof for the case $n_2 = 1$ here.

Proof. Because $n_2 = 1$, $||f(x)||_L$ downgrades to $|f(x)|$. Define $f_{xy}(t) = f(tx + (1-t)y)$ for $0 \le t \le 1$, which is a continuous function. From the definition of $f(x)$, $f_{xy}(t)$ has derivative, and is bounded by a polynomial of $||tx + (1-t)y||_L$. As a result,

$$\begin{aligned} |f(x) - f(y)| &= |f_{xy}(1) - f_{xy}(0)| \\ &= |f'_{xy}(\epsilon)| \\ &\le poly_{Jacobian}(||\epsilon x + (1-\epsilon)y||_L) \\ &\le poly_{Jacobian}(||y||_L + \epsilon||y - x||_L) \\ &= \sum_{i=0}^{n} poly_{Jacobian,i}(||y||_L)||y - x||_L^i \end{aligned} \tag{4}$$

where $poly_{Jacobian}$ represents a polynomial calculated from the Jacobian matrix of f. Because each element is bounded by a polynomial of $||x||_L$, $poly_{Jacobian}$ exists. □

Then, we want to show that some neural network structures are polynomial Lipschitz continuous.

Lemma 3.6. *Some modern widely used neural structures are polynomial Lipschitz continuous functions.*

Proof. We list several widely used neural network-based blocks (in the trained model, not during training) here and show that they are polynomial Lipschitz continuous functions after finishing training.

- Fully Connect Layer: The formula is $Wx + b$, obviously it is a Lipschitz continuous function, which is also a polynomial Lipschitz continuous.
- FCN layer, CNN Layer, Non-linearities (relu, sigmoid, tanh, elu): Under the choices of our norm, they are Lipschitz functions(Kim et al., 2021). Hence, they are polynomial Lipschitz continuous functions.
- Neural Spline Layer: In (Durkan et al., 2019; Cai et al., 2022), they introduce element-wise polynomial layers containing quadratic term and cubic term. For $f(x) = x^n$, its deravative is nx^{n-1}. As a result, by Lemma 3.5 these layers are polynomial Lipschitz continuous functions.
- Attention Layer: We investigate the Jacobian matrix of the multi-head dot-product attention layer in the paper(Kim et al., 2021). In the Jacobian matrix, the element is $J_{ij} = X^T P^{(i)}[E_{ji} X A^T + X A \delta_{ij}] + P_{ij} I$. Although it can be extremely large for large $||X||_{L^p}$, the entry is bounded by a polynomial $||X||_{L^p}$. As a result, Lemma 3.5, is a polynomial Lipschitz function.
- Transformer Layer: It is a combination of matrix multiplication, residual blocks, MLP blocks, Multi-head Attention, and activation functions. As a result, its polynomial Lipschitzable depends on the polynomial Lipschitz continuity of the selected activation functions.

Because there are too many widely used structures, we would not list them all in our main paper. More structures are shown in Appendix A.4 and the proofs are in the same section.

If the trained-well model is the combination of polynomial Lipschitz continuous neural-based blocks, use Lemma 3.3 we can see that the trained-well model is a polynomial Lipschitz continuous function.

□

Now we focus on the property of matrix inversion in a specific area.

Lemma 3.7. *Denote the matrix inversion function as $Inv(x)$. Suppose $A_0 \in \mathbb{R}^{n \times n}$ is a singular matrix with rank $n-1$, $B(A_0, \delta)$ is a ball centered on A_0 with radius ϵ in $\mathbb{R}^{n \times n}$. Denote S_B as the set of all singular matrices in $B(A_0, \delta)$. Then, for any positive number C, we can find a $\delta(C)$ satisfy that for any matrix A in $B(A_0, \delta(C)) \setminus S_B$, $||Inv(x)||_L > C$.*

Proof. We analyse the matrix inversion by the formula

$$A^{-1} = \frac{1}{det(A)} Adj(A) \tag{5}$$

where $Adj(A)$ represents the adjugate matrix, and the $(i,j)-th$ element of $Adj(A)$ is $(-1)^{i+j}$ times the determinant of the $(n-1)\times(n-1)$ matrix that results from deleting row j and column i of A.

Because the determinant of A is the linear combination of the multiplication of specific elements, for any small enough positive number ϵ, $\delta^{det}(\epsilon) = C^{det}_{A_0}\epsilon$ satisfies that for any $A \in B(A_0, \delta^{det}(\epsilon))$, $|det(A) - det(A_0)| < \epsilon$. Here, $C^{det}_{A_0}$ is a constant calculated from A_0. Because A_0 is singular matrix, we have $det(A_0) = 0$. As a result, in $A \in B(A_0, \delta^{det}(\epsilon))$, $|det(A)| < \epsilon$.

Under the $L-$ norm (either L_1, L_2, L_∞), because the element of $Adj(A)$ is the determinant of the sub-matrix, it is easy to prove for any small positive number ϵ, we can find a $\delta^{Adj}(\epsilon)$ satisfies that for any $A \in B(A_0, \delta^{Adj}(\epsilon))$, $|||Adj(A) - Adj(A_0)||_L| < \epsilon$.

We assumed that A_0 is rank $n-1$, which means that $Adj(A_0)$ is not zero-matrix. Hence, $||Adj(A_0)||_L > 0$.

Therefore, we set $\delta(C) = min(\delta^{Adj}(||Adj(A_0)||_L/2), \frac{||Adj(A_0)||_L C^{det}_{A_0}}{2C})$, for any $A \in B(A_0, \delta_\epsilon) \setminus S_B$,

$$\begin{aligned}
||A^{-1}||_L &= \frac{1}{det(A)}||Adj(A)||_L \\
&\geq \frac{1}{det(A)}(||Adj(A_0)||_L - ||Adj(A) - Adj(A_0)||_L) \\
&> \frac{1}{2det(A)}||Adj(A_0)||_L \\
&> \frac{2C}{2||Adj(A_0)||_L}||Adj(A_0)||_L \\
&= C
\end{aligned} \tag{6}$$

□

Now we give the proof of Theorem 3.2.

Proof. Suppose the measure of set M is $m(M)$. We ignore all singular matrices in the set M because the measure of the singular matrix set is 0 and denote it as S_M. From the definition of expectation over Lebsgure measurement,

$$\mathbb{E}_M(||Inv(x) - F(x)||^K_L) = \int_{M\setminus S_M} ||Inv(x) - F(x)||^K_L \frac{1}{m(M)} dm \tag{7}$$

Because $B(\vec{a}, c)$ is contained in the set, which is a ball area and $\vec{a}$ in a data point, c is a sufficiently large number, there should be a singular matrix A_0 with rank $n-1$ with the ball $B(A_0, \epsilon_0) \setminus S_M$ contained in the dataset.

Obviously, $\mathbb{E}_M(||Inv(x) - F(x)||^K_L) \geq \int_{B(A_0,\epsilon_0)\setminus S_M} ||Inv(x) - F(x)||^K_L \frac{1}{m(M)} dm$, we only consider the integral over set $B(A_0, \epsilon_0)$.

Because $F(x)$ is polynomial Lipschitz in $B(A_0, \epsilon_0)$, there must be a maximum value of $||F(x)||_L$, denote it as C_F.

In Lemma 3.7, we find that, for sufficiently small ϵ, in the ball $B(A_0, \epsilon)$, $||Adj(A) - Adj(A_0)||_L < ||Adj(A_0)||_L/2$ and $|det(A)| = |det(A) - det(A_0)| < \frac{\epsilon}{C^{det}_{A_0}}$. We set ϵ small enough to satisfy $\epsilon < \epsilon_0$ and $C_F < \frac{C^{det}_{A_0}||Adj(A_0)||_L}{4\epsilon}$. The last requirements make us conclude that

$$\begin{aligned}
&||\frac{1}{det(A)}Adj(A) - F(x)||_L \\
&\geq \frac{1}{det(A)}||Adj(A)||_L - ||F(x)||_L \\
&\geq \frac{1}{det(A)}(||Adj(A_0)||_L - ||Adj(A) - Adj(A_0)||_L) - ||F(x)||_L \\
&> \frac{1}{2det(A)}||Adj(A_0)||_L - C_F \\
&> \frac{C^{det}_{A_0}}{2\epsilon}||Adj(A_0)||_L - \frac{C^{det}_{A_0}||Adj(A_0)||_L}{4\epsilon} \\
&= \frac{C^{det}_{A_0}}{4\epsilon}||Adj(A_0)||_L > 0
\end{aligned} \tag{8}$$

Hence,

$$\begin{aligned}
&\int_{B(A_0,\epsilon)\setminus S_M} ||Inv(x) - F(x)||^K_L \frac{1}{m(M)} dm \\
&= \int_{B(A_0,\epsilon)\setminus S_M} ||\frac{1}{det(A)}Adj(A) - F(x)||^K_L \frac{1}{m(M)} dm \\
&> \int_{B(A_0,\epsilon)\setminus S_M} (\frac{C^{det}_{A_0}}{4\epsilon}||Adj(A_0)||_L)^K \frac{1}{m(M)} dm \\
&= \frac{1}{m(M)}(\frac{C^{det}_{A_0}||Adj(A_0)||_L}{4})^K \int_{B(A_0,\epsilon)\setminus S_M} (\frac{1}{\epsilon})^K dm \\
&= \frac{1}{m(M)}(\frac{C^{det}_{A_0}||Adj(A_0)||_L}{4})^K C_{n\times n}\epsilon^{n\times n-K},
\end{aligned} \tag{9}$$

where $C_{n\times n}$ is the constant of the measurement of the unit ball $B(0_{n\times n}, 1)$. Because $K > n\times n$, for any positive number E, we can find such a small $\epsilon(E)$ makes $\frac{1}{m(M)}(\frac{C^{det}_{A_0}||Adj(A_0)||_L}{4})^K C_{n\times n}\epsilon(E)^{n\times n-K} > E$. Hence, we have

$$
\begin{aligned}
&\mathbb{E}_M(||Inv(x) - F(x)||_L^K) \\
&= \int_{M \setminus S_M} ||Inv(x) - F(x)||_L^K \frac{1}{m(M)} dm \\
&\geq \int_{B(A_0,\epsilon) \setminus S_M} ||Inv(x) - F(x)||_L^K \frac{1}{m(M)} dm \\
&> \frac{1}{m(M)} (\frac{C_{A_0}^{det} ||Adj(A_0)||_L}{4})^K C_{n\times n} \epsilon^{n\times n - K} \\
&> E.
\end{aligned} \tag{10}
$$

As a result, any trained polynomial Lipschitz continuous neural network-based model $F(x)$ can NOT satisfy $\mathbb{E}_M(||Inv(x) - F(x)||_L^K) < E$ under Lebesgue measurement if $K > n^2$ for any E.

□

In Theorem 3.2, we need K larger than n^2 for a small p in L_p norm. However, in Lemma 3.7, it can be shown that $||Inv(x)||_L$ is extremely huge (can exceed any number precision) around some singular matrices. Because such a set is a non-zero measure in the Lebesgue measure, it is possible to sample such data from the whole space if scientists wish to train a general neural-based end-to-end matrix inversion model. It seems that this statement together with our theorems reveals the reason why there is no general neural-based end-to-end model for matrix inversion in history.

4. Methodology: Where can

In this section, we discuss in which area it is possible to train an end-to-end neural-network-based model to approximate the matrix inversion function. Also, we describe how to design a proper neural-network-based model for matrix inversion.

4.1. Where can

Recall Lemma 3.7, it shows that around singular matrix with $n-1$, $||Inv(x)||_L$ can be extremely large. Hence, if a well-trained neural network-based matrix inversion is established, it should be considered that the training data does not contain any data near all the singular matrices.

Mathematically, suppose that there is a small number ϵ. Define the set M_ϵ as

$$
M_\epsilon = \cup_{A \in S_M} B(A, \epsilon) \tag{11}
$$

where S_M represent the set of all singular matrices, and $B(A, \epsilon)$ is a ball set in $\mathbb{R}^{n\times n}$ centered in A with radius ϵ.

As a result, the training set must not contain M_ϵ if the data precision can not hold enough large numbers.

To make our statement more clear, we give some examples to show the M_ϵ area in 2D/3D figures. In this section, we discuss about the area for the 2×2 matrix inversion problem. Let $\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ denote the 2×2 matrix. Hence, the M_ϵ becomes

$$
M_\epsilon = \cup_{a_{11}a_{22} = a_{12}a_{21}} B(\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \epsilon). \tag{12}
$$

For example, when $a_{11} = 1, a_{12} = 2$ are fixed, the M_ϵ can be constructed by the neighborhood of line $2a_{21} - a_{22} = 0$. The black area in the left figure of Figure 1 shows the M_ϵ set in this case. When $a_{11} = 1, a_{22} = 2$ are fixed, the M_ϵ can be constructed by the neighborhood of curve $a_{12}a_{21} = 2$. The black area in the right figure of Figure 1 shows the M_ϵ set in this case.

When only $a_{11} = 1$ is fixed, we can plot the M_ϵ set in 3D figure. The blue area in Figure 4.1 represents the M_ϵ area, and the yellow ball is an example that can form a possible training dataset for matrix inversion. In this case, if any training data belongs to the blue area, it is impossible to train a neural-based model for 2×2 matrix inversion problem.

4.2. Theoretical Analysis of Model Designing

In this section, we describe how to design the neural network to approximate the matrix inversion problem in a specific area $M_0 = \prod_{i=1,j=1}^{n} [A_{0,i,j} - c, A_{0,i,j} + c] \in \mathbb{R}^{n\times n}$ theoretically. We assume that $M_0 \cap M_\epsilon = \emptyset$.

Let $A_0 + A'$ denote a matrix in set M_0. Obviously, $A' \in \prod_{i=1,j=1}^{n} [-c, c]$. Consider the determinant formula:

$$
det(A) = \sum_{\sigma \in S_n} (sgn(\sigma) \prod_{i=1}^{n} A_{i,\sigma(i)}) \tag{13}
$$

where sgn denotes the permutation function, S_n denotes the symmetric group of all such permutations, $sgn(\sigma)$ is 1 if the permutation can be obtained from even number of exchanges of two entries, -1 otherwise.

It is obviously that

$$
\begin{aligned}
&(A_0 + A')_{k,l}^{-1} = \\
&(-1)^{k+l} \frac{\sum_{\sigma \in S_{n-1}} (sgn(\sigma) \prod_{i=1}^{n-1} (A_{0,i,\sigma(i)}^{adj,k,l} + A_{i,\sigma(i)}^{\prime,adj,k,l}))}{\sum_{\sigma \in S_n} (sgn(\sigma) \prod_{i=1}^{n} (A_{0,i,\sigma(i)} + A'_{i,\sigma(i)}))}
\end{aligned} \tag{14}
$$

where $A_{0,i,\sigma(i)}^{adj,k,l}$ denote the $(i, \sigma(i))$-th element in the matrix that results from deleting row k and column l of A_0.

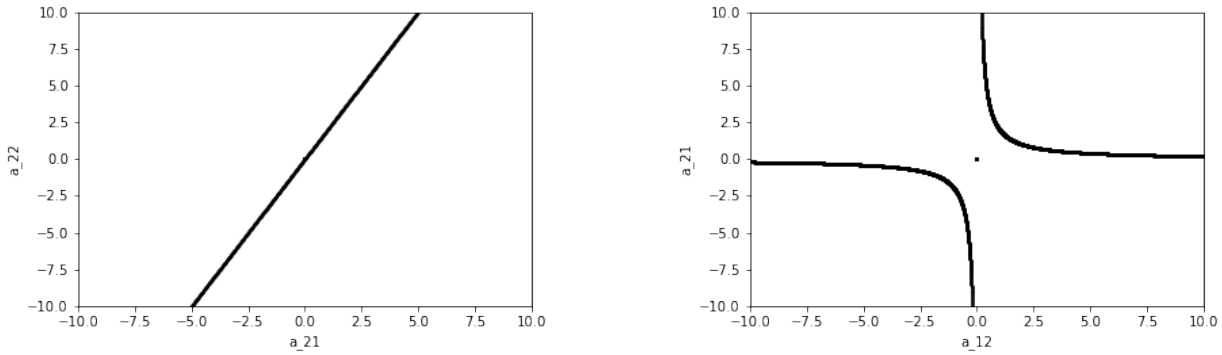

Figure 1. **Left**: 2D M_ϵ area for fixed $a_{11} = 1, a_{12} = 2$ **Right**: 2D M_ϵ area for fixed $a_{11} = 1, a_{22} = 2$

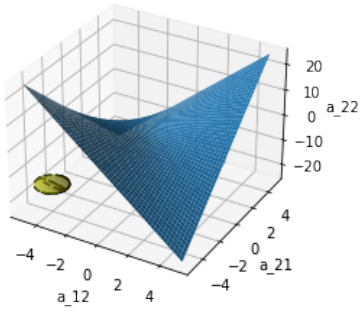

Figure 2. Blue: 3D M_ϵ area for fixed $a_{11} = 1$. Yellow: A possible training dataset area.

Hence, we can show that

$$(A_0+A')^{-1}_{k,l} = f^0_{k,l}(\{A_0\})+\sum_{i,j=1}^{n} f^1_{k,l}(\{A_0\})A'_{i,j}+O(\{A'_{i,j}\}^2) \tag{15}$$

when $A'_{i,j}$ are all small enough. In this formula $f^i_{k,l}(\{A_0\})$ represents a function of all elements in A_0, and $O(\{A'_{i,j}\}^2)$ represents the higher-order term.

As a result, we can design a neural network with 2 fully connected layers to approximate formula 15 as:

$$\begin{aligned} h^{1+}_{k,l} &= ReLU(\sum_{i,j=1}^{n} f^1_{k,l}(\{A_0\})A'_{i,j}) \\ h^{1-}_{k,l} &= ReLU(\sum_{i,j=1}^{n} (-f^1_{k,l}(\{A_0\}))A'_{i,j}) \\ h^2_{k,l} &= f^0_{k,l}(\{A_0\}) + 1 * h^{1+}_{k,l} + 1 * h^{1-}_{k,l} \end{aligned} \tag{16}$$

and the error is $O(\{A'_{i,j}\}^2)$. When A' is small enough, the error term decreases quadratically.

As a result, for the design of a neural-based end-to-end matrix inversion model in a specific area with n^2 input elements, two fully connected layers and one hidden layer with at least $2n^2$ hidden units can perform the linear approximation part.

4.3. Time Complexity Analysis

In the previous section, we discussed the model design. Obviously, the polynomial Lipschitz continuous neural-based model at least need to contain the linear approximation information. Hence, there should be a substructure approximates the linear part in the trained model. We assume that there is at least one substructure like a fully connected layer containing the linear information.

From formula 14 and 15, it is easy to show that the linear approximation part must be dense. That is, $\frac{\partial Inv(A)_{k,l}}{\partial A_{i,j}}$ is independent from each other for different k, l, i, j. Hence, in the fully connected substructure of the model, it should at least contain a $n^2 \times n^2$ full matrix (the input is n^2 elements and output is n^2 elements), which has n^4 computational time complexity in traditional matrix multiplication.

However, the time complexity of the standard matrix inversion algorithm is $O(n^3)$. This time complexity is much smaller than the time complexity of neural networks which can exceed the linear approximation. As a result, this statement may be another reason why currently there is no neural-based general matrix inversion model even in a specific area.

5. Experiments

In this section, we present experiments to support our statement in section 4.2. We perform neural-based models on 4 different matrix inversion datasets and show the results in section 5.2. In Section 5.3, we compare a 2-layer model which has a small number of hidden units with linear approx-

Table 1. Average absolute error on the test set. Numbers in the parenthesis are standard deviations. The average/standard deviation is computed by 3 runs.

Model	2×2(1-st) dataset	2×2(2-nd) dataset
2-Fully Connect	$2.05 \times 10^{-5}(3.78 \times 10^{-11})$	$6.39 \times 10^{-6}(3.72 \times 10^{-12})$
3-Fully Connect	$1.52 \times 10^{-5}(4.31 \times 10^{-11})$	$7.37 \times 10^{-6}(3.95 \times 10^{-12})$
Model	3×3 dataset	16×16 dataset
2-Fully Connect	$8.77 \times 10^{-5}(9.40 \times 10^{-11})$	$1.68 \times 10^{-4}(3.19 \times 10^{-10})$
3-Fully Connect	$1.53 \times 10^{-4}(4.41 \times 10^{-9})$	$2.58 \times 10^{-4}(8.18 \times 10^{-11})$

imation and show what did the models learn experimentally and theoretically.

5.1. Experiment Setup

Datasets

We did the experiments on different sizes of the matrices. To check the parameters in the neural networks and verify the correctness quickly, we select small matrices with sizes 2×2, and 3×3. In order to show our statements work for large matrices, experiments are also conducted on matrices with size 16×16. For 2×2 matrix, in order to show the correctness of our theorem that at any possible neighborhood with no intersection of M_ϵ, we generate two different matrix datasets centered at $\begin{pmatrix} 2 & 2 \\ 2 & 3 \end{pmatrix}$ and $\begin{pmatrix} 2 & 1 \\ 0 & -1 \end{pmatrix}$ seperately, and have the structure like $\prod_{i=1}^{2}\prod_{j=1}^{2}[A_{0,i,j} - 0.01, A_{0,i,j} + 0.01] \in \mathbb{R}^{2\times2}$. The first dataset is set to be centered at a positive definite symmetric matrix, and the second dataset is set to be centered at a more general matrix. We denote them as $2 \times 2(1 - st)$ and $2 \times 2(2 - nd)$ dataset.

For the 3×3 matrix, we generate a dataset centered at $\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 2 & 4 \end{pmatrix}$.

For the 16×16 matrix, we selected a matrix with each element sampled from $\{-2, -1, 0, 1, 2\}$ and checked that it is non-singular. Then the dataset is generated by randomly sampling the matrix around such the matrix.

Implementation Details

We use 2 (or 3) fully connected layers with the ReLU activation function and set the hidden dimensions several times the dimension of the input, and use Adam as the optimizer. The loss is set as the MSE loss. We select hyperparameters after an extensive grid search. To make our results better, the learning rate warm restart technique(Loshchilov & Hutter, 2017) is used during our training. The details of the hyperparameter chosen are shown in Appendix B and Appendix C. Experiments are conducted on either one Nvidia 3080 Laptop GPU locally or one Nvidia V100 GPU on the Google Cloud platform.

5.2. Results

We did several experiments to show that neural networks with 2 or 3 fully connected layers have the ability to learn the approximation of matrix inversion. Table 1 shows the average absolute error between the inverse of the input matrix and the output of the neural network on the test set. For each dataset and each method, 3 experiments were established and the average/standard deviation results were computed from the 3 trials.

Table 1 shows that for 2×2 matrix, the elements in the inversion computed by the neural network are only about 10^{-5} away from the ground truth. This error is small enough to conclude that the neural network has learned the matrix inversion in the space of the dataset. For 3×3 matrix, the error becomes larger to 10^{-4}, and for 16×16 matrix, the average error is about $2 * 10^{-4}$. Although they are larger than the error for the 2×2 matrix, compared with the value in the ground truth inversion matrix, they are small enough.

5.3. Compare with Linear Approximation

In this section, we train a small 2-layer model on the $2 \times 2(1 - st)$ dataset and try to analyze the model theoretically to show what did the small model learned on the matrix inversion problem.

First, we introduce the linear approximation of matrix inversion around $\begin{pmatrix} 2 & 2 \\ 2 & 3 \end{pmatrix}$. Let the input matrix be the form of $\begin{pmatrix} 2 & 2 \\ 2 & 3 \end{pmatrix} + \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, and the inversion has the form of $\begin{pmatrix} 1.5 & -1 \\ -1 & 1 \end{pmatrix} + \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$. From formula 14, we can give

the linear approximation of a_{ij} as

$$\begin{aligned} a_{11} &\approx -2.25a + 1.5b + 1.5c - d \\ a_{12} &\approx 1.5a - 1.5b - c + d \\ a_{21} &\approx 1.5a - b - 1.5c + d \\ a_{22} &\approx -a + b + c - d \end{aligned} \tag{17}$$

Then, we compare the linear approximation method with the neural-network-based method. In Table 2, we compare the average absolute error on the test set for four different models. It is obviously that, neural-network models have better performance than the traditional linear approximation method.

Table 2. Average absolute error on 2×2(1-st) test set for different models. For the deep learning model, the average is computed by 3 runs.

Model	Average absolute error
Linear Approximation	1.97×10^{-4}
2-Fully Connect(small)	6.82×10^{-5}
2-Fully Connect	2.05×10^{-5}
3-Fully Connect	1.52×10^{-5}

Finally, it is needed to discuss the difference between the neural network-based model and linear approximation.

In our experiment, the average absolute distance between the output of the 2-layers model and the linear approximation is 2.3210×10^{-4}, which is almost the same as the error between linear approximation and ground truth shown in Table 2.

Because the neural network can be written as $W_2 ReLU(W_1(a, b, c, d)^T + b_1) + b_2$, we try to compute what the formula represents in different space in $\mathbb{R}^4$. We first trained a 2-layers neural network and fixed the parameters after training. Then, we randomly sampled 1M data points in the dataset area, and found that 55.7% of the sampled data located in the area $\{h_i > 0 | i \in \{1, 4, 5, 6, 7\}\} \cap \{h_i < 0 | i \in \{3\}\}$, where h_i represents the hidden unit in layer 1 of the fixed neural network. If we eliminate the ReLU function for $h_i > 0$ and discard the negative unit, the output a_{ij} of the neural network in this area has the form

$$\begin{aligned} a_{11} =& -2.3034 * a + 1.5408 * b \\ & + 1.5354 * c - 1.0260 * d - 0.0102 \\ a_{12} =& 1.5392 * a - 1.5324 * b \\ & - 1.0302 * c + 1.0241 * d + 0.0081 \\ a_{21} =& 1.5373 * a - 1.0313 * b \\ & - 1.5265 * c + 1.0220 * d + 0.0060 \\ a_{22} =& -1.0290 * a + 1.0248 * b \\ & + 1.0215 * c - 1.0180 * d - 0.0049 \end{aligned} \tag{18}$$

For another 41.7% data, located in $\{h_i > 0 | i \in \{1, 3, 4, 5\}\} \cap \{h_i < 0 | i \in \{6, 7\}\}$, the output a_{ij} has the form

$$\begin{aligned} a_{11} =& -2.1860 * a + 1.4526 * b \\ & + 1.4583 * c - 0.9698 * d - 0.0229 \\ a_{12} =& 1.4544 * a - 1.4624 * b \\ & - 0.9641 * c + 0.9717 * d + 0.0174 \\ a_{21} =& 1.4550 * a - 0.9621 * b \\ & - 1.4679 * c + 0.9733 * d + 0.0167 \\ a_{22} =& -0.9652 * a + 0.9702 * b \\ & + 0.9741 * c - 0.9783 * d - 0.0131 \end{aligned} \tag{19}$$

We can find that the distance between each coefficient in the neural network and each coefficient in the linear approximation formula 17 is smaller than 0.06, which means that they are almost the same as the linear approximation. These two cases include most of the randomly sampled data (97.4%), hence, we can conclude that in most cases of the dataset area, the 2-layers model just learned a better linear approximation of a_{ij}. Appendix D shows the full analysis of all the a_{ij} in all the cases, and shows the property of the neural network for the remaining 2.6% data.

6. Discussion and Conclusions

6.1. Limitations

Due to the lack of computing resources, we did not try the experiments on the large matrix (like $10,000 \times 10,000$) and we did not try a neural network with more layers and units. We also refrained from exploring the performance of non-polynomial Lipschitz networks on the matrix inversion problem.

6.2. Conclusions and future work

In this paper, we have proved that most modern neural network structures can not form a general matrix inversion model. To demonstrate the proofs, we define a generalized Lipschitz function class, which can describe more modern neural network structures. It is interesting to find out the performance of non-polynomial Lipschitz networks on mathematical-related tasks in the future, and how can researchers define a better set to include all the neural network structures to analyze the ability of neural-based methods on their tasks.

We also describe in which area, neural networks can perform well on the matrix inversion problem both theoretically and experimentally. This evidence may reveal what the black-box neural networks learn over specific tasks.

References

Almasadeh, A. J., Alnajjar, K. A., and Albreem, M. A. Enhanced deep learning for massive mimo detection using approximate matrix inversion. In *2022 5th International Conference on Communications, Signal Processing, and their Applications (ICCSPA)*, 2022.

Ben-Shaul, I., Bar, L., Fishelov, D., and Sochen, N. Deep Learning Solution of the Eigenvalue Problem for Differential Operators. *Neural Computation*, 35(6):1100–1134, 05 2023.

Cai, D., Ji, Y., He, H., Ye, Q., and Xi, Y. Autm flow: Atomic unrestricted time machine for monotonic normalizing flows. In *UAI*, 2022.

Dai, J., Chen, Y., Xiao, L., Jia, L., and He, Y. Design and analysis of a hybrid gnn-znn model with a fuzzy adaptive factor for matrix inversion. *IEEE Transactions on Industrial Informatics*, 18(4):2434–2442, 2022.

Dai, J., Tan, P., Xiao, L., Jia, L., He, Y., and Luo, J. A fuzzy adaptive zeroing neural network model with event-triggered control for time-varying matrix inversion. *IEEE Transactions on Fuzzy Systems*, 31(11):3974–3983, 2023.

Durkan, C., Bekasov, A., Murray, I., and Papamakarios, G. Neural spline flows. In *Advances in Neural Information Processing Systems*, pp. 7509–7520, 2019.

Fa-Long, L. and Zheng, B. Neural network approach to computing matrix inversion. *Applied Mathematics and Computation*, 47:109–120, 1992.

Fa-Long, L. and Zheng, B. From zhang neural network to newton iteration for matrix inversion. *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I*, 56(7):1405–1414, 2009.

Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., Ruiz, F. J. R., Schrittwieser, J., Swirszcz, G., Silver, D., Hassabis, D., and Kohli, P. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610: 47–53, 2022.

Gerontitis, D., Mo, C., Stanimirović, P. S., Tzekis, P., and Katsikis, V. N. A novel extended li zeroing neural network for matrix inversion. *Neural Computing and Applications*, 35:14129–14152, 2023.

Han, J., Lu, J., and Zhou, M. Solving high-dimensional eigenvalue problems using deep neural networks: A diffusion monte carlo like approach. *Journal of Computational Physics*, 423:109792, 2020. ISSN 0021-9991.

Jang, J.-S., Lee, S.-Y., and Shin, S.-Y. An optimization network for matrix inversion. In *Neural Information Processing Systems*, 1987.

Kim, H., Papamakarios, G., and Mnih, A. The lipschitz constant of self-attention. In *ICML*, 2021.

Latorre, F., Rolland, P., , and Cevher, V. Lipschitz constant estimation of neural networks via sparse polynomial optimization. In *International Conference on Learning Representations*, 2020.

Li, L. and Hu, J. An efficient second-order neural network model for computing the moore–penrose inverse of matrices. *IET Signal Processing*, 16(9):1106–1117, 2022.

Loshchilov, I. and Hutter, F. SGDR: stochastic gradient descent with warm restarts. In *ICLR*, 2017.

Steriti, R., Coleman, J., and Fiddy, M. A. A neural network based matrix inversion algorithm. In *IJCNN*, 1990.

Virmaux, A. and Scaman, K. Lipschitz regularity of deep neural networks: analysis and efficient estimation. In *Advances in Neural Information Processing Systems*, 2018.

A. Supplement proofs

A.1. Proof of Lemma 3.3

Recall the lemma

Lemma A.1. *Suppose we have two functions $f(x) : \mathbb{R}^{n_2} \to \mathbb{R}^{n_3}$, $g(x) : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}$. And, Under norm L (either L_1, L_2 or L_∞, defined on $\mathbb{R}^{n_i}$), f, g are all polynomial Lipschitz continuous functions. Then the composition $f \circ g$ is also a polynomial Lipschitz continuous function.*

Proof. First we prove that any polynomial Lipschitz continuous function $g(x)$ is bounded by a polynomial of $||x||_{L^1}$ under L^2 norm in $\mathbb{R}^{n_2}$. Recall the definition

$$||f(x) - f(y)||_{L^2} \leq \sum_{i=0}^{n^f} f_i(||x||_{L^1}, ||y||_{L^1})||x - y||_{L^1}^i, \tag{20}$$

we have

$$\begin{aligned} ||g(x)||_{L^2} &\leq ||g(x) - g(0)||_{L^2} + ||g(0)||_{L^2} \\ &\leq \sum_{i=0}^{n^g} g_i(||x||_{L^1}, ||0||_{L^1})||x||_{L^1}^i + ||g(0)||_{L^2} \\ &= \sum_{i=0}^{n^g} g_i(||x||_{L^1}, 0)||x||_{L^1}^i + ||g(0)||_{L^2} \end{aligned} \tag{21}$$

where $g_i(||x||_{L^1}, 0)$ can downgrades to a polynomial with only one variable $||x||_{L^1}$. As a result, $||g(x)||_{L^2}$ is bounded by a polynomial of $||x||_{L^1}$.

Then consider

$$\begin{aligned} ||f(g(x)) - f(g(y))||_L &\leq \sum_{i=0}^{n^f} f_i(||g(x)||_{L^1}, ||g(y)||_{L^1})||g(x) - g(y)||_L^i \\ &\leq \sum_{i=0}^{n^f} f_i(||g(x)||_{L^1}, ||g(y)||_{L^1})(\sum_{j=0}^{n^g} g_j(||x||_{L^1}, ||y||_{L^1})||x - y||_L^j)^i \\ &\leq \sum_{i=0}^{n^f} ploynomial_{f_i}^g(||x||_{L^1}, ||y||_{L^1})(\sum_{j=0}^{n^g} g_j(||x||_{L^1}, ||y||_{L^1})||x - y||_L^j)^i \\ &= \sum_{i=0}^{n^f \times n^g} (f \circ g)_i(||x||_{L^1}, ||y||_{L^1})||x - y||_L^i \end{aligned} \tag{22}$$

where $(f \circ g)_i(\cdot, \cdot)$ can be calculated from $f_i(\cdot, \cdot)$, $g_i(\cdot, \cdot)$ and the upper bound polynomial of $||g||_{L^2}$. The last inequality are from the property of polynomials. □

A.2. Proof of Lemma 3.3

Recall the lemma

Lemma A.2. *Suppose we have two functions $f(x) : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}$, $g(x) : \mathbb{R}^{n_1} \to \mathbb{R}^{n_3}$. And, Under norm L, either L_1, L_2 or L_∞, defined on $\mathbb{R}^{n_i}$, f, g are all polynomial Lipschitz continuous functions. Then the combination $(f, g)(x) : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2+n_3}$ is also a polynomial Lipschitz continuous function.*

Proof.

$$
\begin{aligned}
||(f,g)(x)-(f,g)(y)||_L &= ||(f(x)-f(y),g(x)-g(y))||_L \\
&\leq ||(f(x)-f(y)||_L + ||g(x)-g(y))||_L \\
&\leq \sum_{i=0}^{n^f} f_i(||x||_{L^1},||y||_{L^1})||x-y||_L^i + \sum_{i=0}^{n^g} g_i(||x||_{L^1},||y||_{L^1})||x-y||_L^i \\
&= \sum_{i=0}^{n^{(f,g)}} (f_i+g_i)(||x||_{L^1},||y||_{L^1})||x-y||_L^i.
\end{aligned} \tag{23}
$$

□

A.3. Proof of Lemma 3.5

Recall the Lemma

Lemma A.3. *Suppose function $f(x):\mathbb{R}^{n_1}\to\mathbb{R}^{n_2}$ and Jacobian of f exists everywhere. If the value each element of the Jacobian is bounded by a polynomial of $||x||_L$, f is a polynomial Lipschitz continuous function.*

Proof. Here, L represents either L_1, L_2 or L_∞. We represent $f(x)$ as $f(x)=(f_1(x),f_2(x),...,f_k(x))$, where $f_i(x):\mathbb{R}^{n_1}\to\mathbb{R}$. Define $f_{k,xy}(t)=f_k(tx+(1-t)y)$ for $0\leq t\leq 1$, which is a continuous function. From the definition of $f(x)$, $f_{k,xy}(t)$ has derivative, and is bounded by a polynomial of $||tx+(1-t)y||_L$. As a result,

Then we have

$$
\begin{aligned}
||f(x)-f(y)||_{L^2} &= ||(|f_1(x)-f_1(y)|,|f_2(x)-f_2(y)|,...,|f_k(x)-f_k(y)|)||_{L^2} \\
&= ||(|f_{1,xy}(1)-f_{1,xy}(0)|,|f_{2,xy}(1)-f_{2,xy}(0)|,...,|f_{k,xy}(1)-f_{k,xy}(0)|||_{L^2} \\
&= ||(|f'_{1,xy}(\epsilon_1)|,|f'_{2,xy}(\epsilon_2)|,...,|f'_{k,xy}(\epsilon_k))||_{L^2} \\
&\leq ||(poly_{Jacobian,1}(||\epsilon_1 x+(1-\epsilon_1)y||_{L^1})dt,...,poly_{Jacobian,k}(||\epsilon_k x+(1-\epsilon_k)y||_{L^1})dt)||_{L^2} \\
&\leq ||(poly_{Jacobian,1}(||y||_{L^1}+\epsilon_1||y-x||_{L^1})dt,...,poly_{Jacobian,k}(||y||_{L^1}+\epsilon_k||y-x||_{L^1})dt)||_{L^2} \\
&= ||(\sum_{i=0}^{n_1} poly_{Jacobian,1,i}(||y||_{L^1})||y-x||_{L^1}^i,...,\sum_{i=0}^{n_k} poly_{Jacobian,k,i}(||y||_{L^1})||y-x||_{L^1}^i)||_{L^2} \\
&\leq \sum_{i=0}^{max(\{n_i\})} ||poly_{Jacobian,1,i}(||y||_{L^1}),...,poly_{Jacobian,k,i}(||y||_{L^1})||_{L^2}||y-x||_{L^1}^i \\
&\leq \sum_{i=0}^{max(\{n_i\})} (poly^f_{Jacobian,i}(||y||_{L^1})||y-x||_{L^1}^i
\end{aligned} \tag{24}
$$

□

A.4. Proof of Lemma 3.6

Recall the lemma

Lemma A.4. *Some modern widely used neural structures are polynomial Lipschitz continuous functions.*

Proof. We list the most widely used neural network structures below:

- Fully Connect Layer: The formula is $Wx+b$, obviously, it is a Lipschitz continuous function, which is also a polynomial Lipschitz continuous.
- FCN Layer, CNN Layer, Non-linearities (relu, sigmoid, tanh, elu): Under the choices of our norm, they are Lipschitz functions(Kim et al., 2021). Hence, they are polynomial Lipschitz continuous functions.

- Neural Spline Layer: In (Durkan et al., 2019; Cai et al., 2022), they introduce element-wise polynomial layers containing quadratic term and cubic term. For $f(x) = x^n$, obviously we have

$$\begin{aligned} |f(x) - f(y)| = |x^n - y^n| &= |x - y||\sum_{i=0}^{n-1} x^i y^{n-1-i}| \\ &\leq (\sum_{i=0}^{n-1} |x|^i |y|^{n-1-i})|x - y|, \end{aligned} \tag{25}$$

  As a result, these layers are polynomial Lipschitz continuous functions.

- Residual structure: It has the form of $y = x + f(x)$. If $f(x)$ is polynomial Lipschitz function, satisfying $||f(x_1) - f(x_2)||_{L^2} \leq \sum_{i=0}^{n^f} f_i(||x_1||_{L^1}, ||x_2||_{L^1})||x_1 - x_2||_{L^1}^i$, then $||y_1 - y_2||_{L^2} = ||x_1 - x_2 + f(x_1) - f(x_2)||_{L^2} \leq \sum_{i=0}^{n^f} f_i(||x_1||_{L^1}, ||x_2||_{L^1})||x_1 - x_2||_{L^1}^i + ||x_1 - x_2||_{L^1}$.

- RNN, LSTM Unit: RNN unit is a combination of matrix multiplication, tanh function, and softmax function. Because they are all Lipschitzable, RNN is a Lipschitz function. Because the LSTM unit is a combination of matrix multiplication, tanh function, and activation functions, its polynomial Lipschitzable depends on the polynomial Lipschitz continuity of the used activation functions.

- Attention Layer: We investigate the Jacobian matrix of the multi-head dot-product attention layer in the paper(Kim et al., 2021). In the Jacobian matrix, the element is $J_{ij} = X^T P^{(i)}[E_{ji} X A^T + X A \delta_{ij}] + P_{ij} I$. Although it can be extremely large for large $||X||_{L^p}$, the entry is bounded by a polynomial $||X||_{L^p}$. As a result, it is a polynomial Lipschitz function.

- Transformer Layer: It is a combination of matrix multiplication, residual blocks, MLP blocks, Multi-head Attention, and activation functions. As a result, its polynomial Lipschitzable depends on the polynomial Lipschitz continuity of the selected activation functions.

□

A.5. Proof for evaluation function $\mathbb{E}_M(\frac{||Inv(x)-F(x)||_L^K}{||x||_L^{K'}}) < E$

Theorem A.5. *Suppose the data is sampled from a dataset M in $\mathbb{R}^{n\times n}$, with no singular matrix contained, and $B(\vec{a}, c)$ is contained in the set, which is a ball area and $\vec{a}$ in a data point, c is a sufficiently large number. Then, under either L_1 norm, L_2 norm, or L_∞ norm metric, any trained polynomial Lipschitz continuous neural network-based model $F(x)$ can NOT satisfy $\mathbb{E}_M(\frac{||Inv(x)-F(x)||_L^K}{||x||_L^{K'}}) < E$ under Lebesgue measurement if $K > n^2$ for any E.*

Proof. Suppose the measure of set M is $m(M)$. We ignore all singular matrices in the set M because the measure of the singular matrix set is 0 and denote it as S_M. From the definition of expectation over Lebsgure measurement,

$$\mathbb{E}_M(\frac{||Inv(x) - F(x)||_L^K}{||x||_L^{K'}}) = \int_{M \setminus S_M} \frac{||Inv(x) - F(x)||_L^K}{||x||_L^{K'}} \frac{1}{m(M)} dm \tag{26}$$

Because $B(\vec{a}, c)$ is contained in the set, which is a ball area and $\vec{a}$ in a data point, c is a sufficiently large number, there should be a singular matrix A_0 with rank $n-1$ with the ball $B(A_0, \epsilon_0) \setminus S_M$ contained in the dataset.

Obviously, $\mathbb{E}_M(\frac{||Inv(x)-F(x)||_L^K}{||x||_L^{K'}}) \geq \int_{B(A_0,\epsilon_0)\setminus S_M} \frac{||Inv(x)-F(x)||_L^K}{||x||_L^{K'}} \frac{1}{m(M)} dm$, we only consider the integral over set $B(A_0, \epsilon_0)$.

Because $F(x)$ is polynomial Lipschitz in $B(A_0, \epsilon_0)$, there must be a maximum value of $||F(x)||_L$, denote it as C_F.

In Lemma 3.7, we find that, for sufficiently small ϵ, in the ball $B(A_0, \epsilon)$, $||Adj(A) - Adj(A_0)||_L < ||Adj(A_0)||_L/2$ and $|det(A)| = |det(A) - det(A_0)| < \frac{\epsilon}{C_{A_0}^{det}}$. Because $||A_0||_L > 0$ (otherwise A_0 is a zero matrix, rank is 0 not $n-1$), we

can find a ϵ small enough to make all matrices in $B(A_0, \epsilon)$ satisfy $||x||_L \leq 2||A_0||_L$. Hence, we can set ϵ small enough to satisfy $\epsilon < \epsilon_0$ and $C_F < \frac{C_{A_0}^{det}||Adj(A_0)||_L}{4\epsilon}$, together with $||x||_L \leq 2||A_0||_L$. Then we have

$$\begin{aligned}
&||\frac{1}{det(A)}Adj(A) - F(x)||_L \\
&\geq \frac{1}{det(A)}||Adj(A)||_L - ||F(x)||_L \\
&\geq \frac{1}{det(A)}(||Adj(A_0)||_L - ||Adj(A) - Adj(A_0)||_L) - ||F(x)||_L \\
&> \frac{1}{2det(A)}||Adj(A_0)||_L - C_F \\
&> \frac{C_{A_0}^{det}}{2\epsilon}||Adj(A_0)||_L - \frac{C_{A_0}^{det}||Adj(A_0)||_L}{4\epsilon} \\
&= \frac{C_{A_0}^{det}}{4\epsilon}||Adj(A_0)||_L > 0
\end{aligned} \tag{27}$$

Hence,

$$\begin{aligned}
&\int_{B(A_0,\epsilon)\setminus S_M} \frac{||Inv(x) - F(x)||_L^K}{||x||_L^{K'}} \frac{1}{m(M)} dm \\
&= \int_{B(A_0,\epsilon)\setminus S_M} ||\frac{1}{det(A)}Adj(A) - F(x)||_L^K \frac{1}{||x||_L^{K'}} \frac{1}{m(M)} dm \\
&> \int_{B(A_0,\epsilon)\setminus S_M} (\frac{C_{A_0}^{det}}{4\epsilon}||Adj(A_0)||_L)^K \frac{1}{2^{K'}||A_0||_L^{K'}} \frac{1}{m(M)} dm \\
&= \frac{1}{m(M)}(\frac{C_{A_0}^{det}||Adj(A_0)||_L}{4})^K \frac{1}{2^{K'}||A_0||_L^{K'}} \int_{B(A_0,\epsilon)\setminus S_M} (\frac{1}{\epsilon})^K dm \\
&= \frac{1}{m(M)}(\frac{C_{A_0}^{det}||Adj(A_0)||_L}{4})^K \frac{1}{2^{K'}||A_0||_L^{K'}} C_{n\times n}\epsilon^{n\times n-K},
\end{aligned} \tag{28}$$

where $C_{n\times n}$ is the constant of the measurement of the unit ball $B(0_{n\times n}, 1)$. Because $K > n \times n$, for any positive number E, we can find such a small $\epsilon(E)$ makes $\frac{1}{m(M)}(\frac{C_{A_0}^{det}||Adj(A_0)||_L}{4})^K \frac{1}{2^{K'}||A_0||_L^{K'}} C_{n\times n}\epsilon(E)^{n\times n-K} > E$. Hence, we have

$$\begin{aligned}
&\mathbb{E}_M(\frac{||Inv(x) - F(x)||_L^K}{||x||_L^{K'}}) \\
&= \int_{M\setminus S_M} \frac{||Inv(x) - F(x)||_L^K}{||x||_L^{K'}} \frac{1}{m(M)} dm \\
&\geq \int_{B(A_0,\epsilon)\setminus S_M} \frac{||Inv(x) - F(x)||_L^K}{||x||_L^{K'}} \frac{1}{m(M)} dm \\
&> \frac{1}{m(M)}(\frac{C_{A_0}^{det}||Adj(A_0)||_L}{4})^K \frac{1}{2^{K'}||A_0||_L^{K'}} C_{n\times n}\epsilon^{n\times n-K} \\
&> E.
\end{aligned} \tag{29}$$

As a result, any trained polynomial Lipschitz continuous neural network-based model $F(x)$ can NOT satisfy $\mathbb{E}_M(||Inv(x) - F(x)||_L^K) < E$ under Lebesgue measurement if $K > n^2$ for any E.

□

B. Hyperparameters in Experiment 5.2

- Adam optimizer: learning rate 5e-5, weight decay coefficient 1e-7.

- Warm restart: CosineAnnealingWarmRestarts function in PyTorch, $T\_0 = 3$,$T\_mult = 2$,$eta\_min = 1e-6$.
- Loss function: MSE Loss.
- $2 \times 2(1-st)$ dataset, 2 FC with ReLU:
  - First layer input features 4, output features 32. Second layer input features 32, output features 4.
  - Batch size 128, training data contains 1,000,000 matrices, train 20 epochs. The test set contains 10,000 matrices.
  - Trained for less than 1 hour.
- $2 \times 2(1-st)$ dataset, 3 FC with ReLU:
  - First layer input features 4, output features 32. The second layer input features 32, output features 32. The third layer input features 32, output features 4.
  - Batch size 128, training data contains 1,000,000 matrices, train 20 epochs. The test set contains 10,000 matrices.
  - Trained for less than 1 hour.
- $2 \times 2(2-nd)$ dataset, 2 FC with ReLU:
  - First layer input features 4, output features 32. The second layer input features 32, output features 4.
  - Batch size 128, training data contains 1,000,000 matrices, train 20 epochs. The test set contains 10,000 matrices.
  - Trained for less than 1 hour.
- $2 \times 2(2-nd)$ dataset, 3 FC with ReLU:
  - First layer input features 4, output features 32. The second layer input features 32, output features 32. The third layer input features 32, output features 4.
  - Batch size 128, training data contains 1,000,000 matrices, train 20 epochs. The test set contains 10,000 matrices.
  - Trained for less than 1 hour.
- 3×3 dataset, 2 FC with ReLU:
  - First layer input features 9, output features 72. The second layer input features 72, output features 9.
  - Batch size 128, training data contains 100,000 matrices, train 200,000 steps. The test set contains 10,000 matrices.
  - Trained for less than 1 hour.
- 3×3 dataset, 3 FC with ReLU:
  - First layer input features 9, output features 72. The second layer input features 72, output features 72. The third layer input features 72, output features 9.
  - Batch size 128, training data contains 100,000 matrices, train 200,000 steps. The test set contains 10,000 matrices.
  - Trained for less than 1 hour.
- 16×16 dataset, 2 FC with ReLU:
  - First layer input features 256, output features 2048. The second layer input features 2048, output features 256.
  - Batch size 128, training data are generated during training, train 200,000 steps.
  - Trained for less than 1 hour.
- 16×16 dataset, 3 FC with ReLU:
  - First layer input features 256, output features 2048. The second layer input features 2048, output features 2048. The third layer input features 2048, output features 256.
  - Batch size 128, training data are generated during training, train 200,000 steps.
  - Trained for less than 1 hour.

C. Hyperparameters in Experiment 5.3

- Adam optimizer: learning rate 5e-5, weight decay coefficient 1e-7.
- Warm restart: CosineAnnealingWarmRestarts function in PyTorch, $T\_0 = 3$,$T\_mult = 2$,$eta\_min = 1e-6$.
- Loss function: MSE Loss.
- First layer input features 4, output features 8. The second layer is input feature 8, output features 4.
- Batch size 128, training data contains 1,000,000 matrices, train 20 epochs.
- Trained for less than 1 hour.

D. Full analysis of model trained in Experiment 5.3

In this section, all values are rounded to 5 significant figures. All the double-point precision values, together with codes/pre-trained models can be downloaded from our codebase in the supplement file.

D.1. Parameters of the trained model

In the trained-well model, there are two fully connected layers. Hence, the formula of the neural network can be written as $y = RELU(xW_1^T + b_1)W_2^T + b_2$. We list all the values of the parameters of the trained model below:

- The weight matrix W_1 of 1-st fully connected layer:
  $[0, 0, 0, 0]$
  $[9.5628e-02, 2.9369e-01, 5.3154e-02, -4.4647e-01]$
  $[0, 0, 0, 0]$
  $[1.5360e+00, -1.2078e+00, -7.5969e-01, 6.3958e-01]$
  $[-3.6153e-01, -4.7281e-02, 4.6443e-01, -3.2432e-01]$
  $[3.5525e-01, -1.8960e-01, -6.1029e-01, 1.3732e-01]$
  $[-1.2802e+00, 1.0202e+00, 6.6826e-01, -5.6571e-01]$
  $[-6.5787e-01, 4.8745e-01, 2.9334e-01, -2.2843e-01]$

- The bias b_1 of 1-st fully connected layer:
  $[0, 7.9882e-01, 0, -2.4201e-01, 1.0407e+00, 1.2244e+00, 1.9913e-01, 9.9282e-02]$

- The weight matrix W_2 of 2-nd fully connected layer:
  $[0, -7.5081e-02, 0, -1.1650e+00, 5.8385e-01, -5.0147e-01, 1.1485e+00, 6.6387e-01]$
  $[0, -8.2578e-01, 0, 9.2669e-01, 1.7431e-01, 4.8691e-01, -1.0013e+00, -3.4407e-01]$
  $[0, -4.9190e-01, 0, 5.3894e-01, -8.0408e-01, 1.0796e+00, -6.3829e-01, -1.4140e-01]$
  $[0, 1.0662e+00, 0, -4.5637e-01, 3.7282e-01, -6.5138e-01, 5.4483e-01, 1.0231e-01]$

- The bias b_2 of 2-nd fully connected layer:
  $[-0.23847, 0.12373, 0.055120, -0.56574]$

We define

$$
\begin{aligned}
h_1(a,b,c,d) &= 0.095628 * a + 0.29369 * b + 0.053154 * c - 0.44647 * d + 0.79882 \\
h_3(a,b,c,d) &= 1.5360 * a - 1.2078 * b - 0.75969 * c + 0.63958 * d - 0.24201 \\
h_4(a,b,c,d) &= -0.36153 * a - 0.047281 * b + 0.46443 * c - 0.32432 * d + 1.0407 \\
h_5(a,b,c,d) &= 0.35525e * a - 0.18960 * b - 0.61029 * c + 0.13732 * d + 1.2244 \\
h_6(a,b,c,d) &= -1.2802 * a + 1.0202 * b + 0.66826 * c - 0.56571 * d + 0.19913 \\
h_7(a,b,c,d) &= -0.65787e * a + 0.48745 * b + 0.29334 * c - 0.22843 * d + 0.099282
\end{aligned} \tag{30}
$$

D.2. Analysis of computing a_{ij} in pre-trained neural network

Consider the element a_{ij}, in this network, it can be represented as $a_{ij} = \sum_{k=0}^{7} w_{k,ij} ReLU(h_k(a,b,c,d))$. For example, a_{11} has a form of

$$\begin{aligned} a_{11} = &-(7.5081e-02) * ReLU(h_1) \\ &-(1.1650e+00) * ReLU(h_3) \\ &+(5.8385e-01) * ReLU(h_4) \\ &-(5.0147e-01) * ReLU(h_5) \\ &+(1.1485e+00) * ReLU(h_6) \\ &+(6.6387e-01) * ReLU(h_7) \\ &-0.23847 \end{aligned} \tag{31}$$

We use two methods, random sample by experiments and linear programming, to show that the pre-trained neural network has learned the linear approximation of matrix inversion.

D.2.1. Experiments: Random sample

Recall the dataset area is $\prod_{i=1,j=1}^{2}[A_{0,i,j} - 0.01, A_{0,i,j} + 0.01] \in \mathbb{R}^{2\times 2}$, and $A_0 = \begin{pmatrix} 2 & 2 \\ 2 & 3 \end{pmatrix}$.

We randomly sampled 1M data points in the dataset area and found 55.7 percent of data located in the area $\{h_i > 0 | i \in \{1,4,5,6,7\}\} \cap \{h_i < 0 | i \in \{3\}\}$, and 41.7 percent of data located in the area $\{h_i > 0 | i \in \{1,3,4,5\}\} \cap \{h_i < 0 | i \in \{6,7\}\}$. These two cases include most of the data (97.4%) in the area.

In the first case, we eliminate the ReLU function in formula 31 and get

$$\begin{aligned} a_{11} &= -2.3034 * a + 1.5408 * b + 1.5354 * c - 1.0260 * d - 0.0102 \\ a_{12} &= 1.5392 * a - 1.5324 * b - 1.0302 * c + 1.0241 * d + 0.0081 \\ a_{21} &= 1.5373 * a - 1.0313 * b - 1.5265 * c + 1.0220 * d + 0.0060 \\ a_{22} &= -1.0290 * a + 1.0248 * b + 1.0215 * c - 1.0180 * d - 0.0049 \end{aligned} \tag{32}$$

In the second case, we eliminate the ReLU too:

$$\begin{aligned} a_{11} &= -2.1860 * a + 1.4526 * b + 1.4583 * c - 0.9698 * d - 0.0229 \\ a_{12} &= 1.4544 * a - 1.4624 * b - 0.9641 * c + 0.9717 * d + 0.0174 \\ a_{21} &= 1.4550 * a - 0.9621 * b - 1.4679 * c + 0.9733 * d + 0.0167 \\ a_{22} &= -0.9652 * a + 0.9702 * b + 0.9741 * c - 0.9783 * d - 0.0131 \end{aligned} \tag{33}$$

Compare with the linear approximation

$$\begin{aligned} a_{11} &\approx -2.25a + 1.5b + 1.5c - d \\ a_{12} &\approx 1.5a - 1.5b - c + d \\ a_{21} &\approx 1.5a - b - 1.5c + d \\ a_{22} &\approx -a + b + c - d \end{aligned} \tag{34}$$

, we can find that the distance between each coefficient in the neural network and each coefficient in the linear approximation is smaller than 0.06.

D.2.2. Linear Programming

In this section, we consider that, in each of the area $\cap_k \{(a,b,c,d) | h_k(a,b,c,d) > 0 \; or \; < 0\}$, how large is the distance between the output of the neural network and the linear approximation.

For a_{ij}, this problem can be stated as a linear programming problem:

$$\begin{aligned} &Found\ (a,b,c,d)\ maximize\ |\sum_{k=0}^{7} w_{k,ij} ReLU(h_k(a,b,c,d)) + bias_{ij} - Linear_{ij}(a,b,c,d)| \\ &Subject\ to\ (a,b,c,d) \in [-c,c]^4 \\ &And\ \{h_k > 0 (or h_k < 0)\} \end{aligned} \tag{35}$$

, where the $Linear_{ij}$ function represents

$$\begin{aligned} Linear_{11}(a,b,c,d) &= -2.25a + 1.5b + 1.5c - d \\ Linear_{12}(a,b,c,d) &= 1.5a - 1.5b - c + d \\ Linear_{21}(a,b,c,d) &= 1.5a - b - 1.5c + d \\ Linear_{22}(a,b,c,d) &= -a + b + c - d \end{aligned} \tag{36}$$

For each area, we test linear programming on it. However, many areas, like all the $h_k < 0$, has no overlap between $(a,b,c,d) \in [-c,c]^4$, which is a null-set. As a result, we only test the area that contains data. The results for each a_{ij} are shown in Table 3, 4, 5, 6 separately:

Table 3. Maximum absolute value between the linear approximation of a_{11} and pre-trained neural network in sets which are not null-set.

Data proportion	Area	Maximum absolute value
$< 0.01\%$	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 > 0$	0.00024328
0.26%	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 < 0$	0.00046950
0.88%	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00054643
41.72%	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.0015488
0.06%	$h_1 > 0, h_3 > 0, h_4 < 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.0019683
55.67%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 > 0$	0.0014152
0.92%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 < 0$	0.00046950
0.44%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00054643
0.02%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.00027471
$< 0.01\%$	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 < 0, h_6 > 0, h_7 > 0$	0.0014152
$< 0.01\%$	$h_1 < 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00037102
0.02%	$h_1 < 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.00092748
$< 0.01\%$	$h_1 < 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 > 0$	0.00020061
$< 0.01\%$	$h_1 < 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00037102

Table 4. Maximum absolute value between the linear approximation of a_{12} and pre-trained neural network in sets which are not null-set.

Data proportion	Area	Maximum absolute value
$< 0.01\%$	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 > 0$	0.00018972
0.26%	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 < 0$	0.00031370
0.88%	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00041427
41.72%	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.0012519
0.06%	$h_1 > 0, h_3 > 0, h_4 < 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.0012519
55.67%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 > 0$	0.0011440
0.92%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 < 0$	0.00031370
0.44%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00041427
0.02%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.00021671
$< 0.01\%$	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 < 0, h_6 > 0, h_7 > 0$	0.0011440
$< 0.01\%$	$h_1 < 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00049615
0.02%	$h_1 < 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.0011328
$< 0.01\%$	$h_1 < 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 > 0$	0.00022353
$< 0.01\%$	$h_1 < 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00049615

Table 5. Maximum absolute value between the linear approximation of a_{21} and pre-trained neural network in sets which are not null-set.

Data propotion	Area	Maximum absolute value
$< 0.01\%$	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 > 0$	0.00014555
0.26%	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 < 0$	0.00024849
0.88%	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00020619
41.72%	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.0012065
0.06%	$h_1 > 0, h_3 > 0, h_4 < 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.0017523
55.67%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 > 0$	0.0010813
0.92%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 < 0$	0.00016381
0.44%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00024714
0.02%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.00016381
$< 0.01\%$	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 < 0, h_6 > 0, h_7 > 0$	0.0010813
$< 0.01\%$	$h_1 < 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00029226
0.02%	$h_1 < 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.00079305
$< 0.01\%$	$h_1 < 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 > 0$	0.00019002
$< 0.01\%$	$h_1 < 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00025328

Table 6. Maximum absolute value between the linear approximation of a_{22} and pre-trained neural network in sets which are not null-set.

Data proportion	Area	Maximum absolute value
$< 0.01\%$	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 > 0$	0.00011774
0.26%	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 < 0$	0.00021507
0.88%	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00017554
41.72%	$h_1 > 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.00095610
0.06%	$h_1 > 0, h_3 > 0, h_4 < 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.0012230
55.67%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 > 0$	0.00085934
0.92%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 < 0$	0.00013245
0.44%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00021283
0.02%	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.00013245
$< 0.01\%$	$h_1 > 0, h_3 < 0, h_4 > 0, h_5 < 0, h_6 > 0, h_7 > 0$	0.00085934
$< 0.01\%$	$h_1 < 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00054635
0.02%	$h_1 < 0, h_3 > 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 < 0$	0.0012563
$< 0.01\%$	$h_1 < 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 > 0, h_7 > 0$	0.00016273
$< 0.01\%$	$h_1 < 0, h_3 < 0, h_4 > 0, h_5 > 0, h_6 < 0, h_7 > 0$	0.00042429