Deep Learning Approximation of Matrix Functions: From Feedforward Neural Networks to Transformers

Rahul Padmanabhan

A Thesis
in
The Department
of
Mathematics and Statistics

Presented in Partial Fulfillment of the Requirements
for the Degree of
Master of Science (Mathematics) at
Concordia University
Montréal, Québec, Canada

March 2025

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Rahul Padmanabhan
Entitled: Deep Learning Approximation of Matrix Functions: From Feedforward Neural Networks to Transformers

and submitted in partial fulfillment of the requirements for the degree of

Master of Science (Mathematics)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ External Examiner
Dr. Frederic Godin

_____ Supervisor
Dr. Simone Brugiapaglia

Approved by _____
Dr. Cody Hyndman
Chair of Department of Mathematics and Statistics

Date: March 21, 2025

_____
Dr. Pascale Sicotte
Dean of Faculty of Arts and Science

Abstract

Deep Learning Approximation of Matrix Functions: From Feedforward Neural Networks to Transformers

Rahul Padmanabhan

Deep Neural Networks (DNNs) have been at the forefront of Artificial Intelligence (AI) over the last decade. Transformers, a type of DNN, have revolutionized Natural Language Processing (NLP) through models like ChatGPT, Llama and more recently, Deepseek. While transformers are used mostly in NLP tasks, their potential for advanced numerical computations remains largely unexplored. This presents opportunities in areas like surrogate modeling and raises fundamental questions about AI's mathematical capabilities.

We investigate the use of transformers for approximating matrix functions, which are mappings that extend scalar functions to matrices. These functions are ubiquitous in scientific applications, from continuous-time Markov chains (matrix exponential) to stability analysis of dynamical systems (matrix sign function). Our work makes two main contributions. First, we prove theoretical bounds on the depth and width requirements for ReLU DNNs to approximate the matrix exponential. Second, we use transformers with encoded matrix data to approximate general matrix functions and compare their performance to feedforward DNNs. Through extensive numerical experiments, we demonstrate that the choice of matrix encoding scheme significantly impacts transformer performance. Our results show strong accuracy in approximating the matrix sign function, suggesting transformers' potential for advanced mathematical computations.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Simone Brugiapaglia, for his invaluable guidance and support throughout this thesis. His expertise, kindness and encouragement have been instrumental in shaping this work.

I would also like to thank Dr. Adrian Iovita and Dr. Jean-Philippe Lessard for introducing me to the subject of mathematics, without which this thesis would not have been possible.

I am also grateful to Dr. Lina Fajardo-Gomez, for her immeasurable patience and support throughout my journey in the domain of mathematics.

From a non-mathematical perspective, I would like to thank Floyd Mayweather Jr. for inspiring me. Coming from a humble background, he exemplifies the power of hard work and dedication to one's craft. His ability to remain composed in highly stressful situations is a testament to his mental fortitude—something I deeply admire and strive to achieve.

Finally, I would like to thank Dr. Frederic Godin for serving as the external examiner of this thesis and for his valuable feedback.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In a short span of time, *Artificial Intelligence* (AI) has revolutionized the way we interact with technology. From facial recognition to autonomous vehicles, AI has found applications in various domains. Alan Turing, in 1950, asked the question, *"Can machines think?"* and the question still remains to be answered [102]. *Deep learning* [64] is a subfield of AI that has found applications in various domains such as natural language processing [52, 79, 95], computer vision [62, 63], and speech recognition [29, 75, 106]. Deep learning models are also referred to as *neural networks* or *Deep Neural Networks* (DNNs). From a mathematical standpoint, simplistically speaking, deep learning neural networks are high dimensional functions that are able to learn from data. Better learning usually means that the model is able to learn more complex patterns in the data and this is usually achieved by increasing the number of parameters in the model. What had stopped, or rather, slowed down deep learning models from being able to learn more complex patterns was the lack of computational power to process large amounts of data in parallel. GPUs came to the rescue and were able to process data in parallel and this allowed for the development of deep learning models. Due to performance constraints, there is a need to find ways to improve the performance of these models. Moreover, there is a need to find the optimal architecture for a given task.

Several architectures have been developed for neural networks but the most successful architecture has been the *transformer* [103]. Primarily used in *Natural Language Processing* (NLP), Transformers have advanced NLP by enabling the development of *Large Language Models* (LLMs) such as ChatGPT [77], Llama [101], Claude [8], Deepseek [28] and others [3, 87]. The term GPT itself stands for *Generative Pre-trained Transformer*, highlighting the importance of transformers in these advancements. Mathematically, transformers blocks are mathematically represented as a parameterized function $f_\theta : \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times d}$, where n is the number of tokens in the input sequence and d is the dimension of the token embeddings, for a given set of parameters θ.

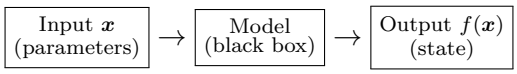
The original transformer model [103] for machine translation was proposed by Vaswani (2017). Additional research and development has given rise to the more recent models like BERT [30] by Devlin (2018), RoBERTa [67] by Liu et al. (2019), GPT-3 [21] by Brown et al. (2020) and Llama-3 [32] by Dubey et al. (2024). Several studies have been conducted to understand the underlying mathematical base of transformers, notably by Charton (2022) with one of the goals of optimizing them [23]. While being revolutionary in the field of NLP, transformers have not been explored to the same depth in the context of advanced numerical computations. There have been studies that have explored the potential of transformers in

this area, but the field is still in its infancy. It has been shown that transformers can perform certain mathematical operations with high degree of accuracy such as, certain problems in linear algebra [25], solve differential equations [60], find the greatest common divisor [24] etc. The error analysis in [25] provided an interesting result which is referred to in Charton (2022) (see [23]). The error analysis of the diagonalization of a matrix task by a transformer shows that, even for answers from transformers which have been determined to be erroneous, transformers have been shown to retain mathematical properties of solutions, such as, generating an incorrect matrix but with the correct eigenvalues. Specifically in the case of the diagonalization of a matrix, in the error cases, the matrix containing the eigenvectors has unit norm within a tolerance of 1% relative error and the eigenvalues are within a tolerance of 1% relative error. This indicates that it learns the latent structure, meaning, very close to the correct eigenvalues and unit norm mutually orthogonal eigenvectors in the case of the diagonalization of a matrix (see [23]).

1.1 Motivation

Our main motivation for this work is to determine the potential of transformers in surrogate modeling. Surrogate modeling is the process of approximating a complex model with a simpler model. High parameter models are difficult to train and are computationally expensive to evaluate. To solve this problem, there is research being done to use transformers in surrogate modeling. FactFormer [66] by Li et al. (2024) have shown that transformers are a promising approach to surrogate modeling of Partial Differential Equations (PDEs). Transformers have been successfully applied to boundary value problems. A notable example is the Galerkin Transformer [22] developed by Cao (2021), which adapts the attention mechanism by removing the softmax function to enable effective operator learning.

Let us begin by describing the general problem of approximating multivariate functions. Given a function $f(\boldsymbol{x})$, where $\boldsymbol{x} \in \mathbb{R}^d$, we aim to approximate the function $f(\boldsymbol{x})$ from a set of pointwise samples $f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_m)$. Such functions typically represent parametric models describing physical processes, which can be conceptualized as:



This framework has broad applications in uncertainty quantification and surrogate modeling across scientific domains [1, 94]. For example, it enables sophisticated weather and climate modeling by capturing complex atmospheric dynamics. In epidemiology, these models help predict disease spread patterns and evaluate intervention strategies. The framework also advances our understanding of subsurface flows and geological formations through detailed hydrological modeling. Nuclear engineering relies on these computational approaches for reactor design optimization and safety analysis. Additionally, biological systems modeling leverages this framework to study cellular processes, protein interactions, and other complex biological phenomena. Matrix functions play a crucial role in many of these applications, particularly in areas requiring sophisticated numerical computations. By developing better ways to approximate matrix functions using transformers, we can potentially improve our ability to handle these complex parametric models more efficiently.

However, approximating such parametric models presents at least four challenges [1]. For one, complex models are high-dimensional, $\boldsymbol{x} = (x_1, \ldots, x_d)$, with $d \gg 1$ or $d = \infty$ which

brings the *curse of dimensionality* [13, 42]. Two, generating data is expensive, it might depend on simulations or physical experiments. Three, the data is corrupted by (unknown) errors, physical errors, discretization error, numerical error, etc. Four, $f(\boldsymbol{x})$ might lie in a Banach or Hilbert space (i.e. models may be function-space valued).

In order to address these challenges, we will use transformers to approximate the function $f(\boldsymbol{x})$. We will apply transformers to approximate a function $\hat{f}(\boldsymbol{x})$ which is a surrogate model for the function $f(\boldsymbol{x})$. There is a trade-off between the accuracy of the surrogate model and the complexity of the model.

The potential of transformers in numerical computations is another motivating factor for this work. While initially seeming suboptimal due to computing performance, we would have to encode the input data prior to using the transformer. This brings the question of what kind of encoding schemes are best for the transformer to use. We aim to study the impact of different encoding schemes on the performance of the transformer. Another motivating factor is that the core of the transformer architecture that is used today, is very similar to what it was from the original paper [103]. While there have been several improvements to the transformer architecture (see [56, 104]), they still require a lot of computational power and memory. If we can understand the limitations of transformers in tasks which are in a different domain than NLP, then in the long term, we can use this knowledge to improve the overall architecture of transformers.

In this thesis, we will analyze *functions of matrices* which falls under the category of high dimensional functions. Functions of matrices are scalar functions extended to matrices. We restrict our attention to functions $f : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$, where n is the dimension of the matrix. A notable example of a function of matrices is the matrix exponential, which is an extension of the exponential function to matrices (and is the most studied per [44]). Statisticians are likely to have come across it in the study of continuous time Markov chains, in the expression $P(t) = e^{tQ}$, where $P(t)$ is the transition probability matrix at time t and Q is the rate matrix. Another important application of matrix functions appears in control theory, where the matrix sign function is used to solve algebraic Riccati equations (AREs) [88] and Lyapunov equations through the computation of stable and unstable invariant subspaces. Matrix functions also have applications in various fields such as physics, engineering, and computer science (see [31, 33, 93]). In this work, we analyze the matrix exponential, matrix logarithm, matrix sign, matrix sine, and matrix cosine. We do not aim to develop a new algorithm to approximate the five matrix functions we are analyzing because the existing algorithms are at a high state of efficiency. We aim to use different neural network architectures and compare their performance to the transformer with different encoding schemes. While it is not possible for us to analyze every single network architecture and every possible combination, we compare two architectures of basic feedforward ReLU [74] networks to the transformer approach.

1.2 Main Contributions

This thesis provides the following two main contributions. The first contribution is a numerical analysis of four methods to approximate the five different functions of matrices which approximates the matrix sign function successfully. The second contribution is a theoretical analysis of the width and depth bounds for a ReLU network to approximate the matrix exponential function.

Numerical Analysis of Matrix Functions using Transformers

The first contribution is a numerical analysis of four methods to approximate the five different functions of matrices from Section 1.1 by learning from randomized data, where, we determine that the transformer encoder-decoder with certain encodings can approximate the matrix sign function (which is not smooth), upto two significant digits, at an accuracy of 88.21% for 3×3 matrices and an accuracy of 82.31% for 5×5 matrices within a tolerance of 1% relative error.

In this analysis, we first attempt to approximate five matrix functions using ReLU [74] neural networks with two different architectures, a neural network with 3 hidden layers and a neural network with 7 hidden layers. We then try to only use the transformer encoder to approximate the same matrix functions. However, literature from Lee et al. (2023) (see [65]) states that an approach with raw input data does not work well. Initial basic tests confirmed this. To attempt to overcome this challenge, we use *Fourier encodings* to enhance the performance of the transformer encoder similar to the way in which Fourier encodings are used in multi-layer perceptrons [99]. This approach by Tancik et al. (2020) was found to be effective for multi-layer perceptrons and helped them to learn high frequency functions in low dimensional domains. Our final approach is to use a transformer encoder-decoder to approximate the matrix functions using four different encoding schemes. The final approach with the transformer encoder-decoder was able to approximate the matrix sign function as listed in the paragraph above. It was based on the following steps. We represent floating point numbers with a sign bit, an exponent, and a mantissa. We use four different encoding schemes to encode the floating point numbers. Each symbol used to represent the number is considered to be a *word* in our dictionary. Each word is represented by an embedding vector also commonly referred to a word embedding. We then construct matrices with coefficents of up to two numerical digits and generate the corresponding matrix function values and train the transformer to approximate the function. We generate random matrices from a normal distribution and bound their coefficents. Existing algorithms are used to compute the corresponding matrix function which is used as the ground truth. The random matrices and the functional result is encoded and is used to train our model.

Theoretical Analysis

Our second contribution is a theorem bounding the width and depth of a ReLU network to approximate the matrix exponential function. The main tools used to prove the theorem are the Taylor series expansion of the matrix exponential function and the approximation of the product of n numbers using ReLU [74] neural networks. We find that the width of the network scales exponentially in nM and the depth scales linearly in M and n. Our proof of the theorem is in Section 4.3.

1.3 Thesis Outline

The thesis is structured as follows. Chapter 2 introduces the theoretical background on matrix functions and some important properties of them. Chapter 3 contains an introduction to transformers, the attention mechanism, and the use of transformers in the context of mathematical computations. Chapter 4 presents our theoretical result on the width and depth bounds for a ReLU network to approximate the matrix exponential function. Chapter 5 lists the experiments we conducted to approximate the matrix functions using shallow

neural networks, deep neural networks, the transformer encoder with Fourier embeddings, and the transformer encoder-decoder. Chapter 6 provides the conclusion of the thesis and outlines possible directions for future work.

Chapter 2

Theoretical background on functions of matrices

This chapter aims to introduce the reader to functions of matrices. The reader may refer to Higham (2008) (see [44]) for more in depth details on the theory of matrix functions. We use the notation of Higham (2008) (see [44]) in this chapter and several of the examples are taken from the book or are adapted from the book.

2.1 Introduction to Matrix Functions

Functions of matrices, also known as matrix functions, are scalar functions extended to matrices. Scalar functions are functions that take a scalar input and return a scalar output, mathematically represented as $f : \mathbb{C} \to \mathbb{C}$. The reader is likely to be familar with several element-wise functions that can be applied to matrices, such as the determinant and trace. However, our focus is on functions that are not element-wise. We want to define a function $f : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$, but we do not want to do it element-wise. For example, consider a matrix $A \in \mathbb{C}^{n\times n}$ and the scalar function $f(x) = \frac{1+x}{1+2x+2x^2}, \quad x \in \mathbb{C}$. For matrices A such that $1 + 2A + 2A^2$ is invertible, we define:

Replacing, x by A, we get

$$f(A) = \frac{1 + A}{1 + 2A + 2A^2}.$$

$$\text{Let} \quad R_1 = (1 + A) \quad \text{and} \quad R_2 = (1 + 2A + 2A^2)^{-1}.$$

$$f(A) = R_1(A)R_2(A) = R_2(A)R_1(A) = (1 + A)(1 + 2A + 2A^2)^{-1}.$$

The result above is from the fact that A commutes with itself and as R_1 and R_2 are rational functions. Hence, we can apply the commutative property of rational functions to matrices. $f(A)$ is called a matrix function. The reader familiar with Padé approximants [11] will recognize that this is in the form of a Padé approximant and one of our definitions of matrix functions will use polynomials to define matrix functions.

The chapter is structured as follows. We start with Section 2.2 which first introduces the three definitions of matrix functions. Section 2.3.1 discusses the properties of matrix

functions. Section 2.4 illustrates the Fréchet derivative. Finally, Section 2.5 presents the five types of matrix functions that we will be analyzing in more detail.

2.2 Matrix Functions Definitions

Matrix functions are defined in three ways:

1. Jordan canonical form
2. Polynomial interpolation
3. Cauchy integral formula

We will now discuss each of these definitions in more detail. Section 2.2.1 discusses the Jordan canonical form. Section 2.2.4 discusses the polynomial interpolation. Section 2.2.6 discusses the Cauchy integral formula.

2.2.1 Jordan Canonical Form

The Jordan canonical form is used to define matrix functions from a theoretical standpoint. While not as numerically stable as the Schur decomposition [71], it is still a useful tool for defining matrix functions, especially for non-diagonalizable matrices. Any square matrix can be represented in Jordan canonical form. For example, the matrix A can be expressed as $A = ZJZ^{-1}$, where J is a Jordan matrix and Z is a non-singular matrix. If A is diagonalizable, then Z is the matrix of eigenvectors of A, and $A = ZDZ^{-1}$, which is the eigen decomposition of A, where D is a diagonal matrix. Let $\lambda_1, \lambda_2, \ldots, \lambda_s$ be the distinct eigenvalues of A. The matrix J is a block diagonal matrix with Jordan blocks on the diagonal, defined as $J = \text{diag}(J_1, J_2, \ldots, J_p)$, where each J_i is a Jordan block associated with some eigenvalue λ_k $(k = 1, 2, \ldots, s)$, and p is the total number of Jordan blocks. By similarity transformation properties (refer to Appendix A.0.2), we can compute $f(A)$ as $f(A) = Zf(J)Z^{-1}$. Thus, we have $Z^{-1}AZ = J = \text{diag}(J_1, J_2, \ldots, J_p)$. However, if A is not diagonalizable, then J is not diagonal, and we need to define $f(J)$. We will provide a definition for $f(J_k)$. Assuming

$$J_k = J_k(\lambda_k) = \begin{bmatrix} \lambda_k & 1 & & \\ & \lambda_k & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_k \end{bmatrix} \in \mathbb{C}^{m_k \times m_k},$$

$f(J_k)$ is defined as

$$f(J_k) := \begin{bmatrix} f(\lambda_k) & f'(\lambda_k) & \cdots & \frac{f^{(m_k-1)}(\lambda_k)}{(m_k-1)!} \\ & f(\lambda_k) & \ddots & \vdots \\ & & \ddots & f'(\lambda_k) \\ & & & f(\lambda_k) \end{bmatrix}.$$

Using this definition, we can compute $f(A)$ as

$$f(A) = Zf(J)Z^{-1} = Z\text{diag}(f(J_1), f(J_2), \ldots, f(J_p))Z^{-1}.$$

We formalize the Jordan canonical form in the following definition [44].

Definition 2.2.1 (Jordan Canonical Form)**.** *Let f be defined on the spectrum of $A \in \mathbb{C}^{n\times n}$ and let A have the Jordan canonical form:*

$$A = ZJZ^{-1} = Z\,\mathrm{diag}(J_1, J_2, \ldots, J_p)Z^{-1}.$$

Then

$$f(A) := Zf(J)Z^{-1} = Z\,\mathrm{diag}(f(J_1), f(J_2), \ldots, f(J_p))Z^{-1},$$

where

$$f(J_k) := \begin{bmatrix} f(\lambda_k) & f'(\lambda_k) & \cdots & \frac{f^{(m_k-1)}(\lambda_k)}{(m_k-1)!} \\ & f(\lambda_k) & \ddots & \vdots \\ & & \ddots & f'(\lambda_k) \\ & & & f(\lambda_k) \end{bmatrix} \quad \text{for } k = 1, 2, \ldots, p.$$

Example 2.2.2 (Matrix Exponential of a Non-diagonalizable Matrix)**.** *Let us take the example of a simple 2×2 matrix A with the goal to compute A^3. This is similar to the example given in Higham (2008) (see [44]). In the scalar case, $f(x) = x^3$. Let*

$$A = \begin{bmatrix} 4 & 3 \\ 7 & 5 \end{bmatrix}.$$

The eigenvalues of A can be verified to be $\lambda_1 = -1$ and $\lambda_2 = 10$. The eigenvectors are

$$v_1 = \begin{bmatrix} 1 \\ -3 \end{bmatrix}, \quad v_2 = \begin{bmatrix} 3 \\ 7 \end{bmatrix}.$$

Therefore, $A = ZDZ^{-1}$ where

$$Z = \begin{bmatrix} 1 & 3 \\ -3 & 7 \end{bmatrix}, \quad D = \begin{bmatrix} -1 & 0 \\ 0 & 10 \end{bmatrix}.$$

Now, $A^3 = ZD^3Z^{-1}$ where

$$D^3 = \begin{bmatrix} -1 & 0 \\ 0 & 1000 \end{bmatrix}.$$

Computing Z^{-1} and multiplying, we obtain

$$A^3 = \begin{bmatrix} 364 & 273 \\ 637 & 455 \end{bmatrix}.$$

We note that there are some assumptions that are made about the function f in the Jordan canonical form:

1. f is defined on the eigenvalues of A;
2. f is differentiable on the eigenvalues of A at least $m - 1$ times.

2.2.1.1 Finiteness of $f(A)$

We can show that $f(A)$ is finite [45]. To do this, let us write $J_k = \lambda_k I + E_k \in \mathbb{C}^{m_k \times m_k}$.

For $m_k = 3$ we have

$$E_k = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \quad E_k^2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad E_k^3 = 0.$$

Assuming f has a Taylor expansion:

$$f(x) = f(\lambda_k) + f'(\lambda_k)(x - \lambda_k) + \cdots + \frac{f^{(j)}(\lambda_k)(x - \lambda_k)^j}{j!} + \cdots.$$

then

$$f(J_k) = f(\lambda_k) I + f'(\lambda_k) E_k + \cdots + \frac{f^{(m_k-1)}(\lambda_k) E_k^{m_k-1}}{(m_k - 1)!}$$

Since E_k is nilpotent (i.e., $E_k^{m_k} = 0$), the series terminates after a finite number of terms. Thus, $f(J_k)$ is a finite matrix, and by extension, $f(A)$ is finite.

We can verify this result with the following example.

Example 2.2.3 (Matrix Exponential of a Non-diagonalizable Matrix)**.** *Let us consider the following non-diagonalizable matrix A and compute e^A*

$$A = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}.$$

This matrix has a repeated eigenvalue $\lambda = 2$ but only one linearly independent eigenvector. Its Jordan canonical form is given by

$$J = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}, \quad Z = I.$$

Therefore, $A = ZJZ^{-1} = J$. To compute e^A, we need to compute e^J:

$$e^J = \begin{bmatrix} e^2 & e^2 \cdot 1 \\ 0 & e^2 \end{bmatrix}.$$

To verify this result, we can compute e^J directly using the series expansion:

$$e^J = \sum_{k=0}^{\infty} \frac{J^k}{k!}.$$

To compute this efficiently, we observe that $J^2 = \begin{bmatrix} 4 & 4 \\ 0 & 4 \end{bmatrix}$, and higher powers of J follow a similar pattern:

$$J^k = \begin{bmatrix} 2^k & k \cdot 2^{k-1} \\ 0 & 2^k \end{bmatrix}.$$

Plugging this into the series for e^J:

$$e^J = \sum_{k=0}^{\infty} \frac{1}{k!} \begin{bmatrix} 2^k & k \cdot 2^{k-1} \\ 0 & 2^k \end{bmatrix}.$$

The elements of e^J become

- *(Top-left):* $\sum_{k=0}^{\infty} \frac{2^k}{k!} = e^2$
- *(Bottom-right):* $\sum_{k=0}^{\infty} \frac{2^k}{k!} = e^2$
- *(Top-right):* $\sum_{k=1}^{\infty} \frac{k \cdot 2^{k-1}}{k!} = e^2$

Thus

$$e^J = \begin{bmatrix} e^2 & e^2 \\ 0 & e^2 \end{bmatrix}.$$

2.2.2 Polynomial Interpolation

Another approach to computing matrix functions is through polynomial interpolation. The Hermite interpolation method provides a way to construct matrix functions by interpolating the function values and their derivatives at the eigenvalues. We use the following definition [44].

Definition 2.2.4 (Matrix Function via Hermite Interpolation)**.** *Let f be defined on the spectrum of $A \in \mathbb{C}^{n \times n}$ and let ψ be the minimal polynomial of A. Then $f(A) := p(A)$, where p is the polynomial of degree less than*

$$\sum_{i=1}^{s} n_i = \deg \psi$$

that satisfies the interpolation conditions

$$p^{(j)}(\lambda_i) = f^{(j)}(\lambda_i), \quad j = 0 : n_i - 1, \quad i = 1 : s$$

where n_i is the algebraic multiplicity of the eigenvalue λ_i and s is the number of distinct eigenvalues. There is a unique polynomial $p(x)$ of minimal degree that satisfies specific conditions, known as the Hermite interpolating polynomial.

This polynomial matches the function values $f(\lambda)$ at each eigenvalue λ and, for eigenvalues with higher multiplicity, also matches the derivatives $f^{(k)}(\lambda)$ up to $m-1$. This approach is particularly useful for non-diagonalizable matrices because it accounts for both eigenvalues and their algebraic multiplicities.

Example 2.2.5 (Matrix Exponential of a Non-diagonalizable Matrix using Polynomial Interpolation)**.** *For instance, consider the same non-diagonalizable matrix in the previous section:*

$$A = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}.$$

The eigenvalue $\lambda = 2$ has multiplicity 2, and the minimal polynomial is $\psi(x) = (x-2)^2$. To compute e^A, we construct a Hermite interpolating polynomial $p(x)$ that satisfies:

$$p(2) = e^2, \quad p'(2) = f'(2) = e^2.$$

This ensures that the matrix function $p(A)$ captures the behavior of e^x at $\lambda = 2$. Substituting $p(A)$ gives the same result as the Jordan canonical form method.

2.2.3 Cauchy Integral Formula

The Cauchy integral formula provides an elegant and powerful way to define matrix functions. We elaborate less on this definition however, the reader can refer to Higham (2008) for more details [44].

Definition 2.2.6 (Matrix Function via Cauchy Integral Formula)**.** *For a matrix $A \in \mathbb{C}^{n\times n}$, we can express a matrix function $f(A)$ as a contour integral:*

$$f(A) = \frac{1}{2\pi i} \oint_\Gamma f(z)(zI - A)^{-1}\, dz,$$

where Γ is a closed contour in the complex plane that encloses all eigenvalues of A, and f is analytic inside and on Γ.

The matrix-valued function $(zI-A)^{-1}$ is called the resolvent of A. It is to be noted that this approach works for any matrix A regardless of its Jordan structure, and for any function f that is analytic in a region containing the spectrum of A.

2.3 Properties of Matrix Functions

Matrix functions have several important properties that make them useful in applications. The following theorem summarizes some key properties.

Theorem 2.3.1 (Higham 2008[44])**.** *Let $A \in \mathbb{C}^{n\times n}$ and let f be defined on the spectrum of A. Then*

1. *$f(A)$ commutes with A;*
2. *$f(A^T) = f(A)^T$;*
3. *$f(XAX^{-1}) = Xf(A)X^{-1}$;*
4. *The eigenvalues of $f(A)$ are $f(\lambda_i)$, where the λ_i are the eigenvalues of A;*
5. *if X commutes with A then X commutes with $f(A)$;*
6. *if $A = (A_{ij})$ is block triangular then $F = f(A)$ is block triangular with the same block structure as A, and $F_{ii} = f(A_{ii})$;*
7. *if $A = diag(A_{11}, A_{22}, \ldots, A_{mm})$ is block diagonal then $f(A) = diag(f(A_{11}), f(A_{22}), \ldots, f(A_{mm}))$;*
8. *$f(I_m \otimes A) = I_m \otimes f(A)$, where $\otimes$ is the Kronecker product;*
9. *$f(A \otimes I_n) = f(A) \otimes I_n$.*

Example 2.3.2 (Matrix Functions Respect Similarity Transformations)**.** *Consider the matrix $A = \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix}$ and $f(x) = e^x$. The eigenvalues of A are $\lambda_1 = 1$ and $\lambda_2 = 3$. By*

property (d), we know that the eigenvalues of e^A must be e^1 and e^3. Furthermore, if we consider a similarity transformation X such that $B = XAX^{-1}$, then by property (c)

$$e^B = e^{XAX^{-1}} = Xe^AX^{-1}$$

This shows that matrix functions respect similarity transformations, which is crucial for numerical implementations.

Properties Similar to Scalar Functions

Matrix functions are required to have properties that are similar to the properties of scalar functions. This approach was first proposed by Fantappie 1928 (see [36]).

The following properties are fundamental to matrix functions:

I. If $f(z) = k$ for some scalar $k \in \mathbb{C}$, then $f(A) = kI$,

II. If $f(z) = z$ for some scalar $z \in \mathbb{C}$, then $f(A) = A$,

III. If $f(z) = g(z) + h(z)$ for some scalar functions g and h, then $f(A) = g(A) + h(A)$,

IV. If $f(z) = g(z) \cdot h(z)$ for some scalar functions g and h, then $f(A) = g(A)h(A)$,

These properties ensure that matrix functions behave consistently with their scalar counterparts.

2.4 Fréchet Derivative

Like in the scalar case, matrix functions can be differentiated. Let U and V be Banach spaces. In general, the Fréchet derivative is a linear mapping from U to V[44].

Definition 2.4.1 (Fréchet Derivative)**.** *The Fréchet derivative of a matrix function $f : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ at a point $X \in \mathbb{C}^{n\times n}$ is a linear mapping*

$$L : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$$
$$E \mapsto L(X, E)$$

such that for all $E \in \mathbb{C}^{n\times n}$

$$\|f(X + E) - f(X) - L(X, E)\| = o(\|E\|),$$

where, $\|.\|$ is a given matrix norm and $o(\|E\|)$ represents a term that goes to zero faster than $\|E\|$ as $\|E\| \to 0$.[1]

This is denoted as the Fréchet derivative at X in the direction of E.

Proposition 2.4.2 (Uniqueness of the Fréchet Derivative)**.** *The Fréchet derivative, if exists, is unique.*

[1]Namely, for $h : \mathbb{C}^{n\times n} \to \mathbb{C}$, we say that $h(E) = o(\|E\|)$ as $\|E\| \to 0$ if $\lim_{\|E\|\to 0} \frac{h(E)}{\|E\|} = 0$.

Proof. Suppose L_1 and L_2 are two Fréchet derivatives of f at X. Then for all $E \in \mathbb{C}^{n\times n}$

$$\|f(X+E) - f(X) - L_1(X,E)\| = o(\|E\|),$$

$$\|f(X+E) - f(X) - L_2(X,E)\| = o(\|E\|).$$

Subtracting the two equations and using the triangle inequality, we get

$$\begin{aligned}\|L_1(X,E) - L_2(X,E)\| &= \|f(X+E) - f(X) - L_1(X,E) - (f(X+E) + f(X) + L_2(X,E))\|,\\ &\leq \|f(X+E) - f(X) - L_1(X,E)\| + \|f(X+E) - f(X) - L_2(X,E)\|,\\ &= o(\|E\|).\end{aligned}$$

This implies that $L_1(X,E) = L_2(X,E)$ for all $E \in \mathbb{C}^{n\times n}$, since the map $E \mapsto L_1(X,E) - L_2(X,E)$ is linear. □

2.5 Types of Matrix Functions

In this section, we focus on the five types of matrix functions that are used in this work.

1. Matrix exponential;
2. Matrix logarithm;
3. Matrix sign;
4. Matrix sine;
5. Matrix cosine.

We will go over each of these functions in a little more detail in the following sections. For a comprehensive treatment of the matrix exponential, we refer the reader to Higham (2008) and Horn and Johnson (2012) (see [44, 48]). In this work, section 2.5.1 presents a definition of the matrix exponential. Section 2.5.2 illustrates a definition of the matrix logarithm. Section 2.5.3 contains a definition of the matrix sign. Section 2.5.4 briefly discusses a definition of the matrix sine. Finally, section 2.5.5 provides a definition of the matrix cosine.

2.5.1 Matrix exponential

The matrix exponential is the most common matrix function. Laguerre and Peano are credited with the definition of the matrix exponential by its power series expansion [44].

Definition 2.5.1 (Matrix Exponential)**.** *Let $A \in \mathbb{C}^{n\times n}$ and $f(x) = e^x$. The power series expansion of the matrix exponential is given by*

$$e^A = \sum_{k=0}^{\infty} \frac{A^k}{k!}.$$

Given that the matrix exponential is one of the most common matrix functions, there are several methods to compute it. In 1978 (reprinted in 2003), Moler and Van Loan (see [73]) compiled "Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later" [73] which has a self explanatory title.

More recently in Khoroshikh and Kurbatov, (2023) (see [54]), it has been determined that the matrix exponential can also be approximated by the Laguerre series expansion

$$H_A = \sum_{n=0}^{\infty} S_{n,\tau,\alpha,A} l_{n,\tau}^{\alpha},$$

where $S_{n,\tau,\alpha,A}$ are the matrix coefficients and $l_{n,\tau}^{\alpha}$ represents the Laguerre functions, which are the modified Laguerre polynomials

$$l_{n,\tau}^{\alpha}(t) = \sqrt{\frac{n!}{\Gamma(n+\alpha+1)}} \tau^{\frac{\alpha+1}{2}} t^{\frac{1}{2}} e^{-\tau t/2} L_n^{\alpha}(\tau t), \quad t \geq 0, n = 0, 1, \dots \quad ,$$

and the generalized Laguerre polynomials are defined as

$$L_n^{\alpha}(t) = \frac{t^{-\alpha} e^t}{n!} (t^{n+\alpha} e^{-t})^{(n)}, \quad \alpha > -1, t \geq 0, n = 0, 1, \dots \quad .$$

In the following proposition, we show that the matrix exponential is not injective. This has been stated in Köyü (2021) (see [59]) however, we provide a proof for completeness.

Proposition 2.5.2 (Non-injectivity of the matrix exponential)**.** *The matrix exponential function* $\exp : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ *is not injective.*

Proof. Consider the matrices

$$A = \begin{bmatrix} 0 & 2\pi i \\ 0 & 0 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad .$$

Using the power series expansion

$$e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots ,$$

note that $A^2 = 0$ (nilpotent), so

$$e^A = I + A = \begin{bmatrix} 1 & 2\pi i \\ 0 & 1 \end{bmatrix} \quad .$$

For matrix B,

$$e^B = I + B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I \quad .$$

Now, consider $C = A + 2\pi i I$. Then

$$e^C = e^A \cdot e^{2\pi i I} = e^A \cdot I = e^A \quad .$$

We know that since $e^{2\pi i} = 1$ (from complex analysis), then $e^{2\pi i I} = I$. Therefore, $e^A = e^C$ but $A \neq C$, showing that the matrix exponential is not injective.

□

This non-injectivity is related to the periodicity of the complex exponential function and the fact that matrices do not necessarily commute.

Applications: The matrix exponential plays a fundamental role across various scientific disciplines. In the study of continuous-time Markov chains, it enables the computation of transition probabilities as systems evolve over time [19, 89]. When solving linear differential equations, the matrix exponential provides elegant solutions of the form $x(t) = e^{At}x(0)$ to systems like $\dot{x}(t) = Ax(t)$ [73]. The quantum mechanics community heavily relies on matrix exponentials to express time evolution operators [41]. In control theory, the matrix exponential naturally emerges in the state transition matrices of linear time-invariant systems, making it an indispensable tool for system analysis and design [43].

2.5.2 Matrix Logarithm

The matrix logarithm is the inverse operation of the matrix exponential, mapping a matrix back to its exponent. It transforms a matrix B to A where $B = e^A$, if such an A exists. This section will provide a definition of the matrix logarithm and some of its properties.

Definition 2.5.3 (Matrix Logarithm)**.** *Let $A \in \mathbb{C}^{n\times n}$, then any matrix X such that $e^X = A$ is called a logarithm of A.*

As we have seen in the previous section, the matrix exponential is not injective, so there are multiple logarithms for a given non singular matrix. In this thesis, we assume that A has no eigenvalues on the negative real axis.

Theorem 2.5.4 (Gantmacher [44])**.** *Let $A \in \mathbb{C}^{n\times n}$ be nonsingular with the Jordan canonical form. Then all solutions to $e^X = A$ are given by*

$$X = ZU\,\mathrm{diag}(L_1^{(j_1)}, L_2^{(j_2)}, \ldots, L_p^{(j_p)})U^{-1}Z^{-1},$$

where

$$L_k^{(j_k)} = \log(J_k(\lambda_k)) + 2j_k\pi i I_{m_k},$$

$\log(J_k(\lambda_k))$ denotes with f the principal branch of the logarithm, defined by $\Im(\log(z)) \in (-\pi, \pi]$, j_k is an arbitrary integer, and U is an arbitrary nonsingular matrix that commutes with J.

Without using the matrix exponential, the following theorem provides a way to compute the matrix logarithm.

Theorem 2.5.5 (Richter [44])**.** *For $A \in \mathbb{C}^{n\times n}$ with no eigenvalues on $\mathbb{R}^-$,*

$$\log(A) = \int_0^1 (A - I)[t(A - I) + I]^{-1}dt.$$

The following theorem provides a way to determine if a matrix has a real valued logarithm. This is important because the matrix logarithm is not defined for matrices with negative eigenvalues.

Theorem 2.5.6 (Higham [44])**.** *Let $A \in \mathbb{R}^{n\times n}$ be nonsingular. Then:*

1. *A has a real logarithm if and only if A has an even number of Jordan blocks of each size for every negative eigenvalue.*

2. *If A has any negative eigenvalues, then no logarithm is real.*

Applications: In quantum information theory, the matrix logarithm is used in defining measures such as the relative entropy [41]. In control systems, the matrix logarithm is used to analyze system dynamics and design controllers, specifically in the discretization of continuous-time systems [43]. More recently, it has been used in molecular simulations where it enables the computation of properties of structural defects in silicon crystals at positive temperatures through efficient and accurate gradients of matrix trace-logarithms [100].

2.5.3 Matrix Sign

Let $A \in \mathbb{C}^{n\times n}$ be a matrix with no eigenvalues on the imaginary axis. For a scalar $z \in \mathbb{C}$, the sign function is defined as

$$\operatorname{sign}(z) = \begin{cases} +1 & \text{if } \operatorname{Re}(z) > 0 \\ -1 & \text{if } \operatorname{Re}(z) < 0. \end{cases}$$

Kenney and Laub (1995) (see [53]) provide a constructive definition of the matrix sign function as a limit of the Newton sequence.

Definition 2.5.7 (Matrix Sign - Newton Sequence)**.** *For a matrix $A \in \mathbb{C}^{n\times n}$ with no eigenvalues on the imaginary axis, the matrix sign function can be defined constructively as a limit of the Newton sequence*

$$A_{n+1} = \frac{1}{2}\left(A_n + A_n^{-1}\right), \quad A_0 = A,$$
$$\operatorname{sgn}(A) \equiv \lim_{n\to+\infty} A_n.$$

Another definition of the matrix sign function is given in Higham (2008) (see [44]) based on the Jordan canonical form.

Definition 2.5.8 (Matrix Sign - Jordan Canonical Form)**.** *If $A = ZJZ^{-1}$ is a Jordan canonical form arranged so that $J = \operatorname{diag}(J_1, J_2)$, where the eigenvalues of $J_1 \in \mathbb{C}^{p\times p}$ lie in the open left half-plane and those of $J_2 \in \mathbb{C}^{q\times q}$ lie in the open right half-plane, then*

$$\operatorname{sign}(A) = Z \begin{bmatrix} -I_p & 0 \\ 0 & I_q \end{bmatrix} Z^{-1}.$$

Applications: The matrix sign function is employed to find solutions to algebraic Riccati equations [88]. The matrix sign function is also used in the solution of Sylvester and Lyapunov equations [96]. It has also been used in model reduction techniques, simplifying complex systems while preserving essential characteristics [17].

2.5.4 Matrix Sine

The matrix sine functions extends the scalar sine function to matrices using its power series expansion. This section will provide a definition of the matrix sine function and some of its properties.

Definition 2.5.9 (Matrix Sine)**.** *The matrix sine function, like its scalar version, can be defined through its power series*

$$\sin(A) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} A^{2k+1}.$$

Euler's formula for the matrix sine function is given by

$$\sin(A) = \frac{e^{iA} - e^{-iA}}{2i}.$$

The matrix sine function also satisfies several important properties analogous to the scalar sine function:

1. $\sin(-A) = -\sin(A)$;
2. If A is real and symmetric, then $\sin(A)$ is real and symmetric;
3. For any invertible matrix S, $\sin(SAS^{-1}) = S\sin(A)S^{-1}$;
4. If A is diagonal, then $\sin(A)$ commutes with A.

Alonso et al. (2017) and Higham and Kandolf (2017) provide efficient algorithms to compute matrix trigonometric functions (see [7, 46]).

Applications: The matrix sine function is used in the solution of second-order differential equations. This has applications in quantum mechanics where the Schrödinger equation is a second-order differential equation which is used to describe the evolution of quantum systems over time [18]. For example, consider the system

$$\frac{d^2}{dt^2} y + A^2 y = 0, \quad y(0) = y_0, \quad y'(0) = y_0',$$

whose solution is given by

$$y(t) = \cos(tA)y_0 + A^{-1}\sin(tA)y_0'.$$

2.5.5 Matrix Cosine

Properties of the matrix cosine function are similar to the matrix sine function.

Definition 2.5.10 (Matrix Cosine)**.** *The matrix cosine function is defined as*

$$\cos(A) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k)!} A^{2k}.$$

Euler's formula for the matrix cosine function is given by

$$\cos(A) = \frac{e^{iA} + e^{-iA}}{2}.$$

When $\|A\| \lesssim 1$, $\cos(A)$ is readily approximated using a Taylor or Padé approximation [44]. Refer to Appendix A.0.3 for the notation $\lesssim$.

Applications: Similar to the matrix sine function, the matrix cosine function is used in the solution of second-order differential equations [44].

Chapter 3

Background on Transformers

This chapter provides a background on the transformer architecture and the mathematical properties of the attention mechanism. In Section 3.2, we provide a brief overview of the transformer architecture and the attention mechanism. In Section 3.2.1, we focus on explaining the attention mechanism in detail. In Section 3.2.1.2, we provide a mathematical representation of transformers. The goal of this chapter is to provide a foundation for how transformers can be applied to matrix computations.

3.1 Overview

A neural network $f(x;\theta)$ is a function that maps an input x to an output y using a set of parameters θ.

$$y = f(x;\theta)$$

While this definition is simple, the complexity of the neural network comes its non-linearity and compositional nature. Taking the case of a highly simple neural network, the neural network learning process begins with initialization by randomly sampling parameter vector θ from a distribution $p(\theta)$. Training then proceeds by minimizing a loss function $L(y, f(x;\theta))$ (refer to Appendix B.0.3). Through backpropagation, gradients are computed and the parameter vector θ is updated to θ^*. Finally, the target function $f(x)$ is approximated by the neural network $f(x;\theta^*)$. A basic neural network is a Feedforward Neural Network (FNN) which is a composition of linear transformations and non-linear activation functions (refer to Appendix B.2). Neural networks have gone through many iterations since the 1950s and several different architectures have been proposed (see [35, 47, 61]). From Convolutional Neural Networks (CNNs) to Recurrent Neural Networks (RNNs), each architecture has its own strengths and weaknesses. For example, CNNs are good at processing images, RNNs are good at processing sequences, and FNNs are good at processing arbitrary data.

The transformer architecture [103] introduced by Vaswani (2017), was a revolutionary architecture that changed the landscape of natural language processing. This architecture was designed to process sequences of data, such as text, in a way that was more efficient than traditional neural networks. Prior to the transformer architecture, a noteable work in sequence-to-sequence modeling was done by Sutskever et al. (2014) with Long Short-Term Memory (LSTM [39]) networks (see [98]). LSTM networks are described in more detail in the appendix (refer to B.4.1 in the appendix). Post 2017, the popularity of the transformer

architecture was apparent by the number of new models that were released based on it, such as BERT [30], RoBERTa [67], GPT-3 [21], and more recently, the Llama family of models [32]. It became a household name with the release of ChatGPT [21] in 2022. It was further popularized by the news worthy release of DeepSeek [28] which caused volatility in the stock market in 2025.

The primary difference between transformers and other neural network architectures is the attention mechanism which replaces the recurrent or convolutional layers used in other architectures. This mechanism allows each element in a sequence to directly interact with every other element, creating a representation of the relationships within the data. As a result, transformers can capture relationships between all elements of a matrix (or sequence) simultaneously through the attention mechanism versus traditional neural networks that process inputs more locally. The attention mechanism which we will go into detail later in this paper, while primarily designed for natural language processing, can be applied to matrix computations. With appropriate design and training, transformers have been shown to solve tasks in Computer Vision [4], Audio Processing [10, 40], and more recently, in mathematical problems [25, 60].

Research by Charton (2021) (see [25]) has shown that transformers can learn to solve problems in linear algebra. The error analysis in this paper has shown that transformers can learn the spectral theorem even when the results are not exact. Taking this a step further, we delve into determining if transformers can learn matrix functions, which can be viewed as a high dimensional function approximation problem. We use the same encodings method as proposed by Charton (2021) to manipulate numeric values by representing them as encodings to generate sequences of encodings.

3.2 Transformer Architecture

The transformer architecture as introduced by Vaswani (2017) and shown in Figure 3.1 consists of several key components:

1. **Self-Attention Layers**: In the context of matrices, these layers enable each matrix element to interact with all other elements, capturing global patterns and being able to learn dependencies. We will go into more detail in Section 3.2.1.

2. **Multi-Head Attention**: In simplistic terms, one can think of multi-head attention as just stacking multiple self-attention layers on top of each other. The computer scientist or programmer may be able to see that processing the multiplication of a vector by a matrix can be done in parallel. The intiution behind this is that, suppose we stack n self-attention layers where each self-attention layer is a function $f_i : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$ called a head. Each layer can theoretically specialize in different aspects where, one head might focus on diagonal elements, another might track block structures, and others could monitor eigenvalue-related patterns. While this may come across as a very simplistic description of multi-head attention, it gives the reader an idea of how this mechanism can be used to process matrices. More detail is provided in Section 3.2.1.1.

3. **Feed-Forward Networks**: In the transformer architecture, the feed-forward network is used to introduce non-linearities to the model. It allows the model to learn complex patterns in the data. Refer to Appendix B.2 for more detail on feed-forward networks.

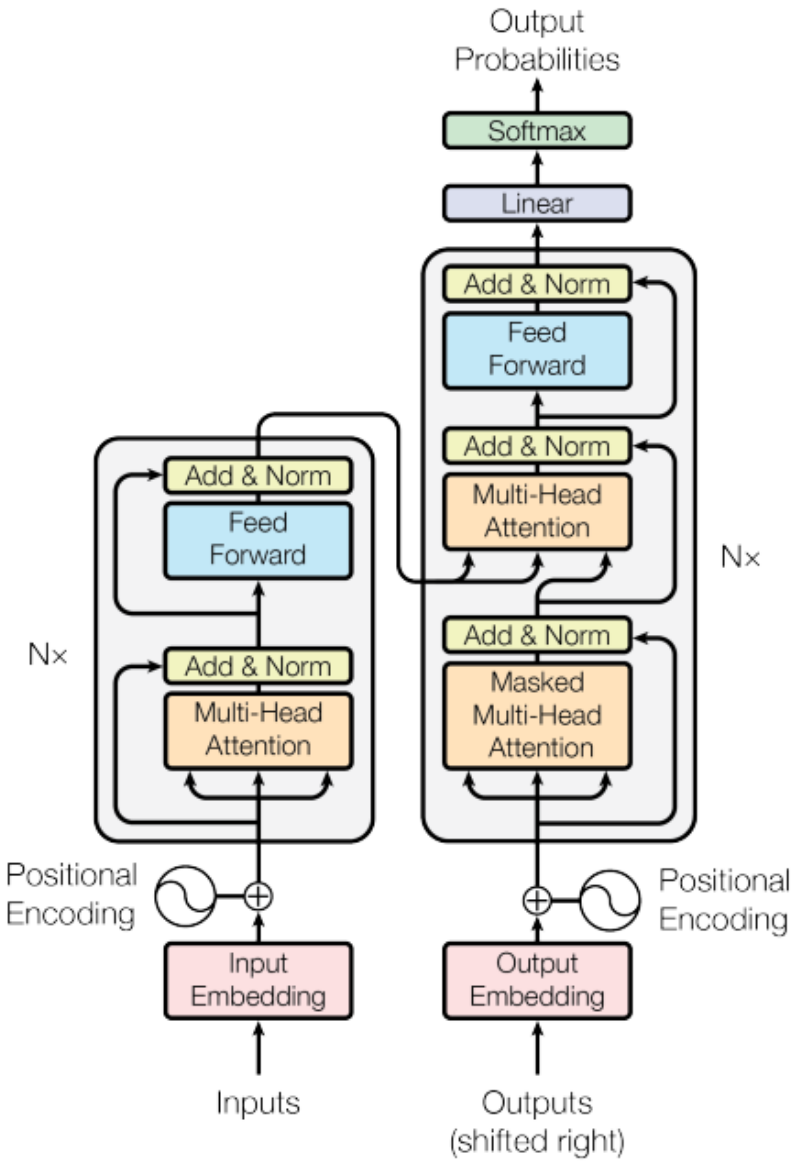


Figure 3.1: The transformer architecture as introduced in Vaswani (2017). The model consists of an encoder (left) and decoder (right) stack. Each encoder layer has a multi-head self-attention mechanism followed by a position-wise feed-forward network. The decoder includes an additional cross-attention layer that attends to the encoder's output. Layer normalization and residual connections are used throughout the architecture to facilitate training and maintain gradient flow. Figure courtesy of [103]

4. **Layer Normalization**: A small, yet significant variation from batch normalization [51], layer normalization normalizes the activations within each individual data sample instead of across a batch. This helps maintain numerical stability when dealing with matrices of varying scales (see Appendix B.1.1 and the original paper by Ba (2016) [9]).

5. **Positional Encoding**: This is most easily understood in the context of NLP where the input is a sequence of words. It helps identify the position of the word and thereby the relevance of the word in the sequence. For matrices, this encodes the row and column positions, helping preserve the 2D structure of the input.

3.2.0.1 Embeddings

Before the transformer can process raw data, if needed, the input matrices must be converted into a format suitable for the attention mechanism. This is accomplished through an embedding process.

For a matrix $A \in \mathbb{R}^{m \times n}$, we define the embedding map:

$$\varepsilon : \mathbb{R}^{m \times n} \to \mathbb{R}^{mn \times d}$$

where d is the embedding dimension. The embedding process consists of three main steps:

1. The input matrix $A \in \mathbb{R}^{m \times n}$ is flattened into a vector in $\mathbb{R}^{mn}$ by concatenating its rows or columns;

2. Each element of this flattened vector is mapped to a d-dimensional embedding vector;

3. These embedding vectors are stacked to form the final embedding matrix of dimensions $mn \times d$.

This embedding transformation projects the scalar matrix elements into a higher-dimensional space where relationships can be better captured. The learned embeddings can encode semantic information about the numerical values. The fixed dimension d provides a consistent input format for the transformer architecture

3.2.1 Attention Mechanism

The attention mechanism is a key component of the transformer architecture which allows the model to focus on the most relevant parts of the input. From an NLP lens and most easily understood to a speaker of any language, suppose we represent a word as a vector, say x. This vector x is one row of a matrix $X \in \mathbb{R}^{n \times d}$ where n is the number of words in the sentence and d is the dimension of the vector space. Now, given this vector $x \in \mathbb{R}^d$, the vector emits a *key*, *query*, and *value*. Keys, queries, and values are mathematically represented as $K, Q, V \in \mathbb{R}^{n \times t}$ where n is the number of words in the sentence and t is the dimension of the transformed space. They are the matrices obtained by projecting the vector x into a higher dimensional space and therefore, $K = XW_k$, $Q = XW_q$, and $V = XW_v$ where $W_k, W_q, W_v \in \mathbb{R}^{d \times t}$ are trainable parameter matrices (in the most basic case). We will use more detailed notation in Section 3.2.1.1 to represent the dimensions of the transformed space. We can then compute the attention scores by applying the softmax function as shown in the equation below. The softmax function applied to the matrix makes

it a stochastic matrix, which can be interpreted as a probability distribution. This attention matrix is given by the formula

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{1}$$

The scaling factor $\sqrt{d_k}$ ensures numerical stability as it prevents the dot product from becoming too large [103].

3.2.1.1 Multi-Head Attention

To capture different types of relationships within the data, transformers use multiple attention heads operating in parallel. Each head can focus on different aspects of the input. Multi-head attention is computed as:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O,$$

where $\text{Concat}(\cdot)$ denotes the concatenation operation that combines multiple matrices along their feature dimension, producing a single matrix of dimension $\mathbb{R}^{n\times hd_v}$ from h matrices of dimension $\mathbb{R}^{n\times d_v}$ The final projection matrix $W^O \in \mathbb{R}^{(hd_v)\times d}$ ensures that the output remains in $\mathbb{R}^{n\times d}$, consistent with the input feature dimension.

Each attention head is computed as

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V),$$

where, $W_i^Q \in \mathbb{R}^{d_{model}\times d_k}$, $W_i^K \in \mathbb{R}^{d_{model}\times d_k}$, $W_i^V \in \mathbb{R}^{d_{model}\times d_v}$ are learned parameter matrices for the i-th head. Here d_{model} represents the dimensionality of the input embeddings.

3.2.1.2 Mathematical Representation

This section provides a mathematical representation of the attention mechanism and multi-head attention. We omit the residual connections for brevity.

Let $H \in \mathbb{N} \setminus \{0\}$ be the number of attention heads and $K \in \mathbb{N} \setminus \{0\}$ be the number of transformer blocks. The single-head attention and multi-head attention mechanisms can be expressed mathematically as

$$\text{Attention}(X) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V,$$

$$\text{Attention}_h(X) = \text{softmax}\left(\frac{Q_hK_h^T}{\sqrt{d_k}}\right)V_h,$$

where for each head $h \in \{1, \ldots, H\}$

$$Q_h = XW_{h,Q}, \quad K_h = XW_{h,K}, \quad V_h = XW_{h,V},$$

with learned parameter matrices

$$W_{h,Q} \in \mathbb{R}^{d_{\text{model}}\times d_k}, \quad W_{h,K} \in \mathbb{R}^{d_{\text{model}}\times d_k}, \quad W_{h,V} \in \mathbb{R}^{d_{\text{model}}\times d_v}.$$

The multi-head attention mechanism concatenates the outputs of all attention heads along the feature dimension

$$\text{MultiHeadAttention}(H, X) = [\text{Attention}_1(X) \mid \text{Attention}_2(X) \mid \cdots \mid \text{Attention}_H(X)].$$

The result of the multi-head attention is then projected using an output matrix $W_o \in \mathbb{R}^{Hd_v \times d_{model}}$:

$$\nabla X = \text{MultiHeadAttention}(H, X)W_o.$$

Here, ∇X represents the result of the multi-head attention projection, not the gradient of X.

A transformer block combines attention with a feed-forward network:

$$\text{TransformerBlock}_i(X) = X + \text{FeedForward}(\text{LayerNorm}(X + \nabla X)) \quad i \in 1, \ldots, K.$$

Finally, the complete transformer is the composition of K transformer blocks:

$$\text{Transformer}(X) = (\text{TransformerBlock}_1 \circ \text{TransformerBlock}_2 \circ \cdots \circ \text{TransformerBlock}_K)(X).$$

3.2.1.3 Cross-Attention

Cross-attention, also known as encoder-decoder attention, is a mechanism that allows the decoder to attend to the encoder's output sequence. Unlike self-attention where queries, keys, and values come from the same sequence, in cross-attention

- Queries (Q) come from the decoder's previous layer;
- Keys (K) and Values (V) come from the encoder's output.

An intuitive explanation of cross-attention is that it enables the decoder to focus on relevant parts of the input sequence when generating each element of the output sequence. In machine translation, cross-attention allows each word in the target language to be generated while considering all words in the source language sentence. Similar to self-attention, the cross-attention mechanism can be represented mathematically as

$$\text{CrossAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Example 3.2.1 (Cross-Attention Example)**.** *Consider a simple sequence of three words: "The owl hoots". Each word is first embedded into a vector space. Let's say we have:*

$$\textit{"The"} \to [1.0, 0.2, -0.3]$$
$$\textit{"owl"} \to [0.4, 1.1, 0.8]$$
$$\textit{"hoots"} \to [-0.2, 0.9, 1.2]$$

The attention mechanism would compute attention scores between each pair of words. For instance, when processing "owl", the model might assign: high attention to "hoots" (as it's the verb describing the owl's action) and medium attention to "The" (as it's the article modifying "owl")
These attention scores are then used to create a weighted combination of the values, producing a contextualized representation for each word that incorporates information from the entire sequence.

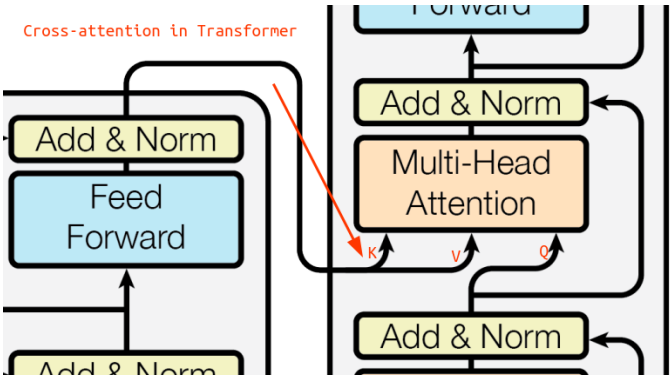


Figure 3.2: Cross-attention mechanism in transformer decoder. The queries come from the decoder while keys and values come from the encoder output. Figure courtesy of [58]

3.2.1.4 Positional Encoding

Since the attention mechanism itself is position-agnostic, positional encoding is used to provide information about the sequence order. The positional encoding is typically added to the input embeddings before they are processed by the transformer. The standard approach uses sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}}),$$
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}}),$$

where pos is the position in the sequence, i is the dimension, and d_{model} is the embedding dimension. This encoding ensures that each position has a unique encoding and the relative positions of tokens can be easily computed.

Example 3.2.2 (Positional Encoding Example)**.** *Returning to our "The owl hoots" example, let's see how positional encoding affects the representation. Suppose we use a simplified 3-dimensional positional encoding:*

$$PE_{pos=1} = [0.84, 0.54, 0.00]$$
$$PE_{pos=2} = [0.41, -0.71, 0.58]$$
$$PE_{pos=3} = [-0.41, -0.71, -0.58]$$

Adding these to our word embeddings:

$$\text{"}The\text{"} + PE_1 = [1.0, 0.2, -0.3] + [0.84, 0.54, 0.00] = [1.84, 0.74, -0.30]$$
$$\text{"}owl\text{"} + PE_2 = [0.4, 1.1, 0.8] + [0.41, -0.71, 0.58] = [0.81, 0.39, 1.38]$$
$$\text{"}hoots\text{"} + PE_3 = [-0.2, 0.9, 1.2] + [-0.41, -0.71, -0.58] = [-0.61, 0.19, 0.62]$$

Now each word's representation contains both semantic information (from the embedding) and positional information (from the encoding). This allows the transformer to mathematically understand that "The" comes before "owl" and "hoots" is the last word.

3.3 Transformers in Mathematics

Since the transformer architecture is a general purpose architecture, it can be applied to a wide range of mathematical tasks. Recent research has demonstrated that transformers can be effectively applied to mathematical tasks, ranging from symbolic mathematics to numerical computations [25, 60]. This application extends beyond simple arithmetic to complex mathematical operations including linear algebra, calculus, and even theorem proving. Existing research by Polu and Sutskever (2020) (see [84]) has shown that a deep learning based system can contribute proofs to the math community Charton (2021) (see [25]) showed certain problems in linear algebra can be solved using transformers. Results from this paper show that transformers can learn to perform fundamental matrix operations such as addition, multiplication, and transposition with high accuracy. They also show that transformers can learn to perform more complex operations such as eigenvalue decomposition and matrix inversion. Polu et al. (2022) (see [85]) showed that expert iteration in the context of language modeling can solve multiple challenging problems from high school olympiads Alfarano et al. (2023) (see [5]) showed that transformers can find Lyapunov functions of polynomial and non-polynomial systems with high accuracy. Charton (2024) (see [24]) showed that transformers can learn the Greatest Common Divisor (GCD) of two numbers. Our work uses the work by Charton (2021) (see [25]) as a foundation.

Recent work by Charton (see [23, 24]) has also made progress in understanding how transformers process mathematical information, providing insights into their internal representations and decision-making processes. This research suggests that transformers can learn to implement known algorithms while also discovering novel computational approaches. The application of transformers to mathematics represents a significant step toward automated mathematical reasoning and computation, though challenges remain in areas such as formal proof verification and handling very large mathematical expressions. A notable work by Alfarano et al. (2024) (see [6]) demonstrates transformers' potential in solving long-standing open problems in mathematics. Their work focuses on finding global Lyapunov functions, a challenging problem in dynamical systems theory that had remained unsolved for decades.

3.4 Limitations

Despite their remarkable success, transformer models face several important limitations. The self-attention mechanism has quadratic computational and memory complexity with respect to sequence length ($O(n^2)$, where n is the sequence length). This makes processing long sequences computationally expensive and memory-intensive. Most transformer implementations have a fixed maximum sequence length (context window) that limits the amount of information they can process at once. This constraint can be problematic for tasks requiring understanding of very long documents or maintaining long-term dependencies. Sanford et al. (2024) have found that transformers are not able to represent certain mathematical concepts. They found that transformers have a complexity that scales linearly with the input size for tasks like triple detection (see [92]). Nogueira et al. (2021) found that transformers have problems with basic arithmetic operations (see [76]). Nogueira et al. also indicates that subword tokenizers and positional encodings are components of the transformer that need improvement. Transformers require large amounts of data to train and this is a problem for domains with limited data availability. The amount of power

required to train a transformer is also significant [69, 82]. To attest to this, some of the experiments in our work have taken several weeks to train. And as with other domains of deep learning, transformer interpretability is a challenge [105]. Various approaches have been proposed to address these limitations, such as sparse attention mechanisms to reduce computational complexity, adaptive positional encodings, and more efficient architectures (see [14, 56, 91, 104]). However, these challenges remain active areas of research in the field.

Chapter 4

DNN Architecture for Matrix Exponential

The theory behind functional approximation using neural networks has a history going back to the 1940s [68]. Kolmogorov in the 1950s proposed a theorem that can be thought of from a neural network perspective [57]. Kolmogorov's superposition theorem [57] states that any continuous function of n variables can be represented as a sum of $2n + 1$ continuous functions of one variable. More precisely, for any continuous function $f : [0, 1]^n \to \mathbb{R}$, there exist continuous functions $\Phi_q : \mathbb{R} \to \mathbb{R}$ and constants λ_{qp} such that

$$f(x_1, x_2, \ldots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \lambda_{qp} x_p \right).$$

If we were to connect it to neural network terms and theoretical aspects, it says that any continuous function of n variables can be represented exactly through a 2-layer neural network of width 2n+1 [34]. While it may not be the foundation of neural networks, it is an interesting historical result that relates to the representational power of neural networks. In the 1980s, Cybenko [27] established the Universal Approximation Theorem, demonstrating that a single hidden layer neural network with a continuous sigmoid activation function can approximate any continuous function on compact subsets of $\mathbb{R}^n$. With AI rising in popularity, there has been a lot of interest in understanding the theoretical bounds of neural networks and a lot of work has been done in this area [12, 15, 34, 70]. In this work, we focus on this specific function rather than a class of functions. The objective of this chapter is to prove a theorem that gives us a depth and width bound of a ReLU neural network that can approximate the matrix exponential. We outline the steps of the proof in Section 4.1, provide the necessary definitions and theorems in Section 4.2, then prove the theorem in Section 4.3.

4.1 Proof Overview

In this chapter, we will prove Theorem 4.3.1. We find the width and depth bounds for a ReLU neural network that can approximate the matrix exponential up to arbitrary accuracy. We then find the error bound for the network. The proof is divided into three steps. The first step is to determine the value of k for the Taylor series for the matrix exponential with

the remainder term. The second step is to find an upper bound for the remainder term. The third step is to construct the network Φ that approximates the matrix exponential.

4.2 Preliminaries

Before proceeding with the proof, we need several key definitions and theorems. First, we recall the definition of the matrix exponential (see [44]).

Definition 4.2.1 (Matrix Exponential)**.** *For a matrix $A \in \mathbb{C}^{n\times n}$, the matrix exponential is defined as*

$$\exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!},$$

where $A^0 = I$ is the identity matrix.

We also need the following lemma about the Taylor series expansion of the matrix exponential (see [44]).

Lemma 4.2.2 (Taylor Series with Remainder for Matrix Exponential)**.** *For any matrix $A \in \mathbb{C}^{n\times n}$ and any non-negative integer K,*

$$\exp(A) = \sum_{k=0}^{K} \frac{A^k}{k!} + R_K(A),$$

where $R_K(A)$ is the remainder term given by

$$R_K(A) = \frac{A^{K+1}}{(K+1)!} \exp(tA),$$

for some $t \in [0, 1]$.

We will also use Stirling's approximation for factorials ([37, Appendix C.12]).

Lemma 4.2.3 (Stirling's Approximation)**.** *For any positive integer n,*

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \exp\left(\frac{\theta(n)}{12n}\right),$$

where $\theta(n)$ is a function taking values in $[0, 1]$.

Finally, we recall some basic properties of matrix norms that will be used in the proof (see [48]).

Lemma 4.2.4 (Matrix Norm Properties)**.** *For matrices $A, B \in \mathbb{C}^{n\times n}$:*

1. $\|AB\|_F \leq \|A\|_F \|B\|_F$,
2. $\|A + B\|_F \leq \|A\|_F + \|B\|_F$,
3. *For any scalar c, $\|cA\|_F = |c|\|A\|_F$.*

We use the following lemma in the proof of Theorem 4.3.1. This is Lemma 7.1 from Adcock et al. (2025) (see [2]).

Lemma 4.2.5 (Approximate Multiplication of l Numbers by ReLU DNNs)**.** *Let $0 < \delta < 1$, $l \in \mathbb{N}$ and consider constants $M_1, \dots, M_l > 0$ such that $M = \prod_{i=1}^{l} M_i \geq 1$. Then there exists a ReLU DNN $\chi_\delta : \prod_{i=1}^{l} [-M_i, M_i] \to \mathbb{R}$ satisfying*

$$\sup_{|x_i| \leq M_i} \left| \prod_{i=1}^{l} x_i - \chi_\delta^{(l)}(\mathbf{x}) \right| \leq \delta, \quad \text{where } \mathbf{x} = (x_i)_{i=1}^{l},$$

The width and depth are bounded by

$$\operatorname{width}(\chi_\delta^{(l)}) \leq c_{1,1} \cdot l,$$

$$\operatorname{depth}(\chi_\delta^{(l)}) \leq c_{1,2} \left(1 + \log(l) \left\lceil \log(l\delta^{-1}) + \log(M) \right\rceil\right),$$

where $M = \prod_{i=1}^{l} M_i$, and $c_{1,1}$, $c_{1,2}$ are universal constants.[1]

We will show the formation for the entries of A^k by induction in the following lemma.

Lemma 4.2.6 (Matrix Multiplication Representation)**.** *Let A be an $n \times n$ matrix. Then, for any $k \in \mathbb{N}$, the entries of A^k are given by*

$$(A^k)_{ij} = \sum_{\ell_1=1}^{n} \sum_{\ell_2=1}^{n} \cdots \sum_{\ell_{k-1}=1}^{n} \prod_{q=1}^{k} a_{\ell_{q-1}\ell_q}, \tag{2}$$

where $\ell_0 = i$ and $\ell_k = j$.

Proof. We will prove the formula by induction.

Base case ($k = 1$):
When $k = 1$, we have $A^1 = A$, and $(A^1)_{ij} = a_{ij}$. In this case, there are no summations and the product has only one term: $a_{\ell_0\ell_1} = a_{ij}$, which matches the matrix entry.

Inductive step:
Assume the formula holds for some positive integer k. We will prove it holds for $k + 1$. Consider $(A^{k+1})_{ij}$. By definition of matrix multiplication

$$(A^{k+1})_{ij} = (A^k \cdot A)_{ij} = \sum_{m=1}^{n} (A^k)_{im} \cdot a_{mj}. \tag{3}$$

Using our inductive hypothesis for $(A^k)_{im}$

$$\begin{aligned}(A^{k+1})_{ij} &= \sum_{m=1}^{n} \left(\sum_{\ell_1=1}^{n} \sum_{\ell_2=1}^{n} \cdots \sum_{\ell_{k-1}=1}^{n} \prod_{q=1}^{k} a_{\ell_{q-1}\ell_q} \right)_{im} \cdot a_{mj}, \\ &= \sum_{m=1}^{n} \sum_{\ell_1=1}^{n} \sum_{\ell_2=1}^{n} \cdots \sum_{\ell_{k-1}=1}^{n} \left(\prod_{q=1}^{k} a_{\ell_{q-1}\ell_q} \right) \cdot a_{mj}.\end{aligned}$$

[1] While [1, Lemma 7.1] does not explicity mention the assumption $M \geq 1$, further analysis and inspection of the proof shows that this assumption is needed.

Now, let $m = \ell_k$. This gives us

$$(A^{k+1})_{ij} = \sum_{\ell_1=1}^{n} \sum_{\ell_2=1}^{n} \cdots \sum_{\ell_k=1}^{n} \prod_{q=1}^{k+1} a_{\ell_{q-1}\ell_q}, \tag{4}$$

where $\ell_0 = i$ and $\ell_{k+1} = j$. This matches our formula for $k+1$, completing the inductive step. By the principle of mathematical induction, the formula holds for all positive integers k. □

These definitions and theorems provide the foundation for constructing and analyzing our neural network approximation of the matrix exponential.

4.3 DNN Architecture for Matrix Exponential

We will prove the Theorem 4.3.1 which gives us the width and depth bounds for a ReLU neural network that can approximate the matrix exponential. We will implicity assume that the neural networks mentioned in the section can accept inputs in the form of a matrix and output a matrix up to reshaping the input and output matrices as vectors.

Theorem 4.3.1 (DNN Architecture for Matrix Exponential)**.** *Let $n \in \mathbb{N}$ and $M \geq 1$. Consider the matrix exponential function $\exp : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$. Then, for any $\epsilon > 0$ there exists a ReLU network f_ϵ with*

$$\begin{aligned}
&\text{width}(f_\epsilon) \leq C_1 \cdot K \cdot n^K, \\
&\text{depth}(f_\epsilon) \leq C_2 \left[1 + \ln(K)\left(\ln(K) + \ln\left(\frac{2\text{e}}{\epsilon}\right) + K\left(\ln(n) + \ln(M)\right)\right)\right],
\end{aligned}$$

that satisfies

$$\sup_{A\in[-M,M]^{n\times n}} \|f_\epsilon(A) - \exp(A)\|_F \leq \epsilon,$$

where $K = \left\lceil \max\left\{ \text{e}nM, \frac{nM + \ln(\frac{\sqrt{2}}{\sqrt{\pi}\epsilon})}{\ln(2)} - 1 \right\} \right\rceil$, *$\|\cdot\|_F$ denotes the Frobenius norm and $C_1, C_2 > 0$ are universal constants.*

Proof. Refer to the auxiliary lemmas in Section 4.2. The matrix exponential can be expressed as

$$\exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!}.$$

Step 1: First, we determine the value of k for the Taylor series for the matrix exponential with the remainder term. The Taylor series for the matrix exponential with the remainder term is given by (see Lemma 4.2.2)

$$\exp(A) = \sum_{k=0}^{K} \frac{A^k}{k!} + \frac{A^{K+1}}{(K+1)!} \exp(tA), \quad \text{for some } t \in [0,1].$$

For any $K \geq 0$, let $R_K(A)$ denote the remainder term

$$R_K(A) = \frac{A^{K+1}}{(K+1)!} \exp(tA), \quad t \in [0,1],$$

Then, by standard matrix norm inequalities (see Lemma 4.2.4),

$$\|R_K(A)\|_F \leq \frac{1}{(K+1)!}\|A\|^{K+1}\|\exp(tA)\|_F.$$

Using the definition of the Frobenius norm, for every $A \in [-M, M]^{n\times n}$, we see that

$$\begin{aligned}\|A\|_F &= \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n}|a_{ij}|^2},\\ &\leq \sqrt{n^2M^2},\\ &= nM.\end{aligned}$$

It follows that $\|A\|_F^{K+1} \leq (nM)^{K+1}$ for any $A \in [-M, M]^{n\times n}$. Therefore,

$$\begin{aligned}\|\exp(tA)\|_F &= \left\|\sum_{j=0}^{\infty}\frac{(tA)^j}{j!}\right\|_F,\\ &\leq \sum_{j\geq 0}\frac{t^j\|A\|_F^j}{j!},\\ &= \exp(t\|A\|_F),\\ &\leq \exp(\|A\|_F) \quad \text{(maximum value at } t = 1),\\ &\leq \exp(nM).\end{aligned}$$

By using Stirling's approximation (refer to Lemma 4.2.3), we can approximate the factorial term and obtain

$$\begin{aligned}\|R_K(A)\|_F, &\leq \frac{(nM)^{K+1}\exp(nM)}{(K+1)!},\\ &= \frac{(nM)^{K+1}\exp(nM)}{\sqrt{2\pi}\cdot(K+1)^{\frac{1}{2}}\cdot(K+1)^{K+1}\cdot e^{-(K+1)}\cdot\exp\left(\frac{\theta(K+1)}{12(K+1)}\right)}, \qquad \text{where } \theta(x) \in [0,1],\\ &\leq \frac{(enM)^{K+1}\exp(nM)}{\sqrt{2\pi}\cdot(K+1)^{K+\frac{3}{2}}\cdot\exp\left(\frac{\theta(K+1)}{12(K+1)}\right)},\\ &\leq \frac{1}{\sqrt{2\pi}}\left(\frac{enM}{K+1}\right)^{K+1}\exp(nM),\\ &\leq \frac{1}{\sqrt{2\pi}}\left(\frac{1}{2}\right)^{K+1}\exp(nM), \qquad (5)\end{aligned}$$

where, in the last step, we assumed that $K \geq 2enM - 1$. Now we find an upper bound for the remainder term.

Step 2: For a given $\eta > 0$, we need to find the value of K such that

$$\|R_K(A)\|_F \leq \eta.$$

Using the upper bound from equation (5)

$$\frac{1}{\sqrt{2\pi}}\left(\frac{1}{2}\right)^{K+1}\exp(nM) \leq \eta,$$

which is equivalent to

$$\frac{\eta\sqrt{2\pi}}{\exp(nM)} \geq \left(\frac{1}{2}\right)^{K+1}.$$

Taking the natural logarithm yields

$$(K+1)\ln\left(\frac{1}{2}\right) \leq \ln\left(\frac{\eta\sqrt{2\pi}}{\exp(nM)}\right)$$

Solving for K, we obtain

$$K \geq \frac{\ln\left(\frac{\exp(nM)}{\eta\sqrt{2\pi}}\right)}{\ln(2)} - 1.$$

It follows that

$$\left\|\exp(A) - \sum_{j=0}^{K} \frac{A^j}{j!}\right\|_F \leq \eta,$$

provided that

$$K \geq \max\left\{enM, \frac{\ln\left(\frac{\exp(nM)}{\sqrt{2\pi}\eta}\right)}{\ln(2)} - 1\right\},$$

which simplifies to

$$K \geq \max\left\{enM, \frac{nM + \ln\left(\frac{1}{\sqrt{2\pi}\eta}\right)}{\ln(2)} - 1\right\}. \tag{4.2}$$

Step 3: We now construct the network Φ that approximates the matrix exponential. Observe that the choice $\eta = \frac{\epsilon}{2}$ guarantees $\|R(A)\|_F \leq \frac{\epsilon}{2}$. Hence, we construct Φ such that

$$\left\|\Phi(A) - \sum_{j=0}^{K} \frac{A^j}{j!}\right\|_F \leq \frac{\epsilon}{2}.$$

In fact, since we have the value of K as a function of the target remainder bound $\eta = \frac{\epsilon}{2}$, using the triangle inequality, this would imply that

$$\|\Phi(A) - \exp(A)\|_F \leq \left\|\Phi(A) - \sum_{j=0}^{K} \frac{A^j}{j!}\right\|_F + \|R(A)\|_F \leq \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon.$$

Now, to construct the network $\mathcal{P}^{(k)}$ that approximates the matrix exponential, i.e., $\sum_{j=0}^{K} \frac{A^j}{j!}$, we will use the Lemma 4.2.6 from Section 4.2, namely,

$$(A^k)_{ij} = \sum_{\ell_1=1}^{n} \sum_{\ell_2=1}^{n} \cdots \sum_{\ell_{k-1}=1}^{n} \prod_{q=1}^{k} a_{\ell_{q-1}\ell_q} \quad \text{where } \ell_0 = i, \ell_k = j.$$

Using Lemma 4.2.5 from Section 4.2, we can construct a ReLU network $\mathcal{P}^{(k)} : \mathbb{R}^{n^2} \to \mathbb{R}^{n^2}$ such that

$$\mathcal{P}^{(k)}(A) \approx A^k \quad \forall A \in [-M, M]^{n\times n}.$$

This network is defined as

$$\mathcal{P}^{(k)}(A)_{ij} = \sum_{\ell_1=1}^{n} \cdots \sum_{\ell_{k-1}=1}^{n} \chi_\delta^{(k)} \left((a_{\ell_{q-1},\ell_q})_{q=1}^{k} \right).$$

This is simply the parallelization of n^{k-1} networks $\chi^{(k)}$, whose outputs are summed up with a final linear layer of 1's. So from Lemma 4.2.5, we have

$$\begin{aligned}
&\text{width}\left(\mathcal{P}^{(k)}\right) = n^{k-1} \cdot \text{width}\left(\chi^{(k)}\right) \leq Cn^{k-1}k, \\
&\text{depth}\left(\mathcal{P}^{(k)}\right) = \text{depth}\left(\chi^{(k)}\right) + 2 \leq C'(1 + \log(k)[\log(k\delta^{-1}) + \log(M^k)]),
\end{aligned}$$

where C and C' are universal constants $C, C' \geq 1$.

Therefore, we define Φ as

$$\Phi(A) = \sum_{j=0}^{K} \frac{1}{j!} \mathcal{P}^{(j)}(A).$$

Now, for any $A \in [-M, M]^{n\times n}$, we have the error bound

$$\begin{aligned}
\left\| \Phi(A) - \sum_{j=0}^{K} \frac{A^j}{j!} \right\|_F &\leq \sum_{j=0}^{K} \frac{1}{j!} \left\| \mathcal{P}^{(j)}(A) - A^j \right\|_F, \\
&= \sum_{j=0}^{K} \frac{1}{j!} \sqrt{\sum_{s=1}^{n}\sum_{t=1}^{n} \left| \sum_{\ell_1=1}^{n} \cdots \sum_{\ell_{j-1}=1}^{n} \left(\left(\chi_\delta^{(k)} \left(a_{\ell_{q-1},\ell_q} \right)_{q=1}^{k} \right) - \prod_{q=1}^{j} a_{\ell_{q-1},\ell_q} \right) \right|^2}, \\
&\leq \sum_{j=0}^{K} \frac{1}{j!} \sqrt{\sum_{s=1}^{n}\sum_{t=1}^{n} \left(\sum_{\ell_1=1}^{n} \cdots \sum_{\ell_{j-1}=1}^{n} \left(\left(\chi_\delta^{(k)} \left(a_{\ell_{q-1},\ell_q} \right)_{q=1}^{k} \right) - \prod_{q=1}^{j} a_{\ell_{q-1},\ell_q} \right) \right)^2}, \\
&\quad \text{(by the Cauchy-Schwarz inequality)}, \\
&\leq \sum_{j=0}^{K} \frac{1}{j!} \sqrt{\sum_{s=1}^{n}\sum_{t=1}^{n} n^{j-1} \sum_{\ell_1=1}^{n} \cdots \sum_{\ell_{j-1}=1}^{n} \delta^2} \quad , \\
&= \sum_{j=0}^{K} \frac{1}{j!} \sqrt{n^2 \cdot n^{j-1} \cdot n^{j-1} \cdot \delta^2} \quad , \\
&= \delta \sum_{j=0}^{K} \frac{n^j}{j!}, \\
&\leq \delta \mathrm{e}^n.
\end{aligned}$$

Therefore, in order to bound the last term by $\frac{\epsilon}{2}$, we let

$$\delta = \frac{\epsilon}{2\mathrm{e}^n}.$$

Recall that $K \geq \max\left\{ enM, \frac{nM + \ln(\frac{1}{\sqrt{2\pi}\eta})}{\ln(2)} - 1 \right\}$. Substituting $\eta = \frac{\epsilon}{2}$ into the expression, we get

$$K \geq \max\left\{ enM, \frac{nM + \ln(\frac{1}{\sqrt{2\pi}\epsilon})}{\ln(2)} - 1 \right\}.$$

We pick the ceilng of the expression, i.e., $K = \left\lceil \max\left\{ enM, \frac{nM+\ln(\frac{\sqrt{2}}{\sqrt{\pi}\epsilon})}{\ln(2)} - 1 \right\} \right\rceil$.
So the final width and depth bounds are:

$$\begin{aligned}
\text{width}(\Phi(A)) &\leq \sum_{j=0}^{K} \text{width}(\mathcal{P}^{(j)}) \leq \sum_{j=0}^{K} \hat{C}_1 n^{j-1} j, \\
&\leq \hat{C}_1 \cdot K \cdot \sum_{j=1}^{K} n^{j-1}, \\
&= \hat{C}_1 \cdot K \cdot \frac{n^K - 1}{n-1}, \\
&\leq D \cdot K \cdot n^K.
\end{aligned}$$

Then let $C_1 = D$. So, we have a power of K in the width bound.

Now, for the depth bound, we have:

$$\begin{aligned}
\text{depth}(\Phi(A)) &\leq \text{depth}(\mathcal{P}^{(K)}) + 2 \\
&\leq C' \left[1 + \ln(K)(\ln(K \cdot \delta^{-1}) + \ln(M^K))\right] + 2 \\
&\leq C' \left[1 + \ln(K) \left(\ln\left(K \cdot \frac{2en^K}{\epsilon} \right) + \ln(M^K) \right)\right] \\
&= C' \left[1 + \ln(K) \left(ln(K) + \ln\left(\frac{2\text{e}}{\epsilon}\right) + K\ln(n) + K\ln(M) \right)\right] \\
&= C' \left[1 + \ln(K) \left(ln(K) + \ln\left(\frac{2\text{e}}{\epsilon}\right) + K\left(\ln(n) + \ln(M)\right) \right)\right]
\end{aligned}$$

These bounds are expressed purely in terms of ϵ, n, and M. Finally, we let $C_1 = D$ and $C_2 = C'$ to get the final width and depth bounds. □

4.4 Discussion

Theorem 4.3.1 shows that the matrix exponential can be approximated with an arbitrary accuracy by a sufficiently wide and deep ReLU neural network. The proof of the theorem is based on an explicit construction of a network realizing an approximate Taylor expansion of the matrix exponential of order K. Up to logarithmic factors, the Taylor expansion order K scales like nM. So, the width scales like $K \cdot n^K \approx Mn \cdot n^{nM}$. Our width bound is hence exponential in nM. However, the depth scales like $\ln(K)\left(\ln(K) + \ln\left(\frac{2\text{e}}{\epsilon}\right) + K\left(\ln(n) + \ln(M)\right)\right)$. So, we have

$$\begin{aligned}
\text{depth} &\approx K\ln(K) \cdot (\text{logarithmic factors}), \\
&\approx Mn \cdot (\text{logarithmic factors}).
\end{aligned}$$

So, up to log factors, the depth scales linearly in M and n. Our proof strategy can probably be optimized and this could lead to improved width and depth bounds. Two possible optimizations are the choice of K and the construction of a network to approximate the matrix power A^k (needed for the Taylor expansion), which could be done in a recursive manner.

Chapter 5

Numerical Experiments

This chapter presents our experiments to approximate the five matrix functions listed in Section 2.5 using different neural network architectures. Section 5.2 explains how to interpret the experimental results, followed by Section 5.3 which describes our experimental setup and methodology. Our investigation progresses through several architectural approaches: a neural network with three hidden layers (Section 5.3.1), a deeper network with seven hidden layers (Section 5.3.2), and transformer-based architectures—first with Fourier embeddings (Section 5.3.4), and then with an encoder-decoder using four different encodings (Section 5.3.5). The latter section contains one of the two main contributions of this thesis.

5.1 Overview

Our goal is to use the transformer architecture to approximate matrix functions. While existing algorithms for matrix function approximation are already efficient [26, 44, 50, 86], our approach is to use the transformer architecture to approximate matrix functions to understand the capabilities of the transformer architecture and the limitations of it. Before we take on the task of approximating matrix functions using transformers, we first ask ourselves the following questions:

1. Can a feedforward neural network be used to approximate matrix functions?
2. What neural network architecture is best suited for this task?
3. Can we just feed the matrix into the transformer without embeddings and use the transformer to approximate the matrix function?

We attempt to answer these questions by conducting experiments with different neural network architectures and encodings. Observe that a matrix function f can be thought of as a function with $n \times n$ parameters (by the reshaping of the matrix). One may then reformulate the problem as a high-dimensional vector valued function approximation problem [1]. Our experiments are designed by varying the number of training samples, the DNN architecture, and the encoding.

5.2 Understanding the Results

We use similar visualizations as mentioned in Adcock et al. (2022) (see Appendix A.1 in this work or [1, Appendix A.1.3]). Our experimental results are visualized through plots with logarithmic scales on both axes. The x-axis displays training sample sizes ranging from 2^5 to 2^{18} samples for experiments detailed in Sections 5.3.1, 5.3.2, and 5.3.4, while the y-axis shows the relative error computed using the Frobenius norm. Shaded regions in these plots represent the dispersion and variability of the error measurements across different trials.
In analyzing these visualizations, several key indicators help assess model performance. Strong performance manifests as relative errors below 10^{-2} (1%) coupled with consistent downward trends as the sample size increases. The stability of predictions can be gauged by the width of the shaded regions, with narrower bands, especially at larger sample sizes, indicating more reliable results. A gradual flattening of the error curve may suggest that additional training data yields diminishing returns.

Conversely, relative errors exceeding 10^{-1} (10%) signal suboptimal performance. Other concerning patterns include error trends that remain flat or exhibit inconsistency despite increased training data, as well as wide shaded regions that indicate high variability in the model's predictions. These characteristics help identify limitations in the model's learning capacity or potential issues in the training process. Due to the high computational cost of training the models, we ran different models on different GPUs. We used several machines which had Nvidia 3080, 4090, and A100 GPUs, however, the computation was not parallelized across the GPUs.

5.3 Experiments

With the goal of approximating the five matrix functions listed in Section 2.5, we use four different types of neural network architectures.

- A relatively shallow neural network[1];
- A deeper neural network[2];
- A transformer encoder with the Fourier Transform by Tancik et al. (2020);
- A transformer encoder-decoder with encodings as listed in Table 5.1.

The following sections detail the configuration of each of the neural network architectures, the parameters of the experiments, the training and testing data, and the results of the experiments.

5.3.1 Shallow Neural Network

To begin our investigation, we first explored whether traditional neural network architectures could effectively approximate the five matrix functions from Section 2.5.
The following is the configuration for the shallow neural network:

- Input layer: Accepts flattened matrices of varying dimensions (1×1 to 8×8)

[1] A neural network with three hidden layers
[2] A neural network with seven hidden layers

- Hidden layers: Three layers with sizes 128, 256, and 128 neurons respectively
- Output layer: Matches input dimension to reconstruct the transformed matrix
- Activation functions: ReLU for hidden layers, linear activation for output layer
- Optimizer: Adam [55] with learning rate 1×10^{-3}
- Batch size: 128
- Training epochs: 100

5.3.1.1 Training and Testing Data

We train $14 \times 5 \times 8 = 560$ models. 14 training set sizes ranging from 2^5 to 2^{18} samples and for each of the five matrix functions and each of the eight matrix dimensions. We use a test set of 2^{15} samples in all cases. The training and testing data is generated by sampling the matrix coefficients from a uniform distribution with coefficients in $[-1, 1]$.

5.3.1.2 Results

The results for the shallow neural network configuration are shown in Figures 5.1 through 5.8. These figures illustrate the performance across different matrix dimensions from 1×1 to 8×8 matrices. We see that the performance of the shallow neural network is not good for any of the matrix dimensions over the trivial case of 1×1 matrix. Note that the variance of the sign function is very high, which is expected as the sign function is not smooth. The sign function has the highest relative error of all the matrix functions in most cases with the exception of the 1×1 matrix.

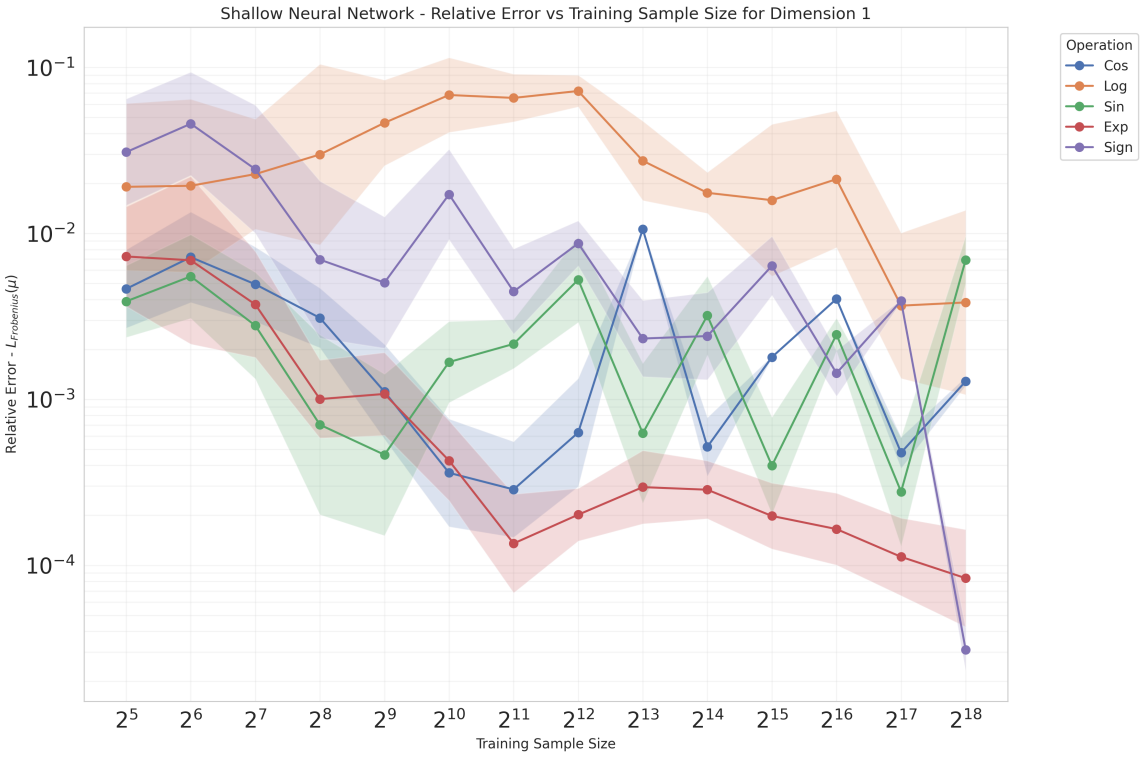


Figure 5.1: Shallow neural network results for 1×1 matrices

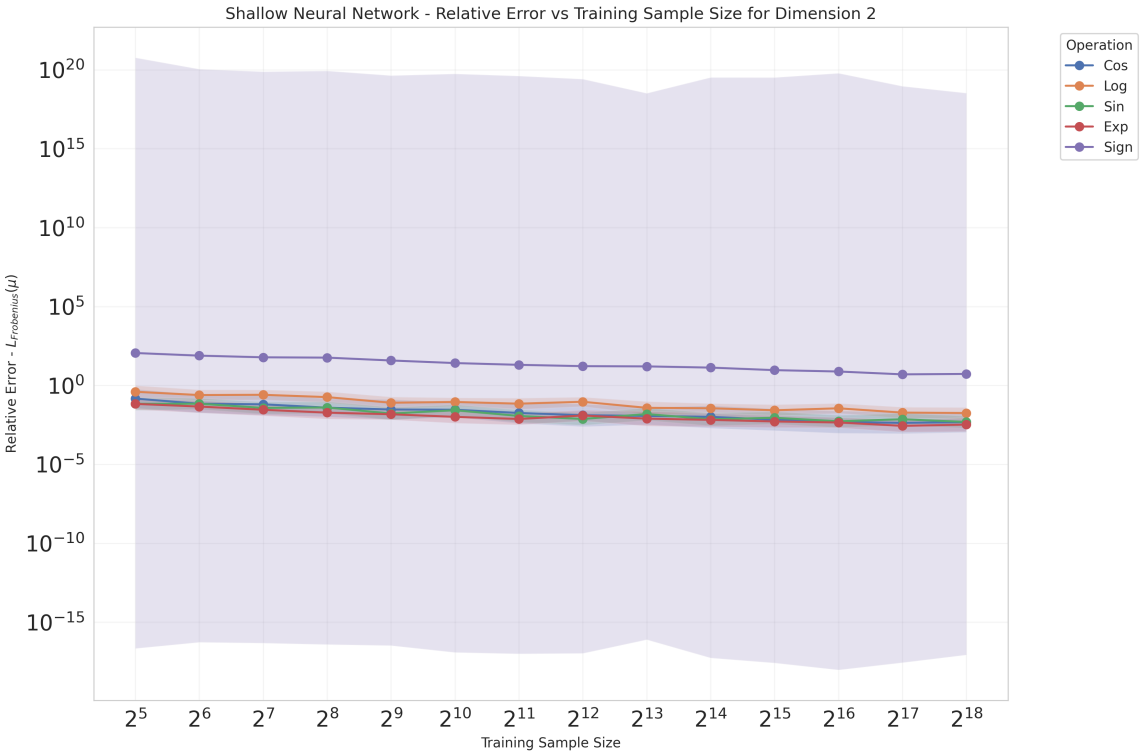


Figure 5.2: Shallow neural network results for 2×2 matrices

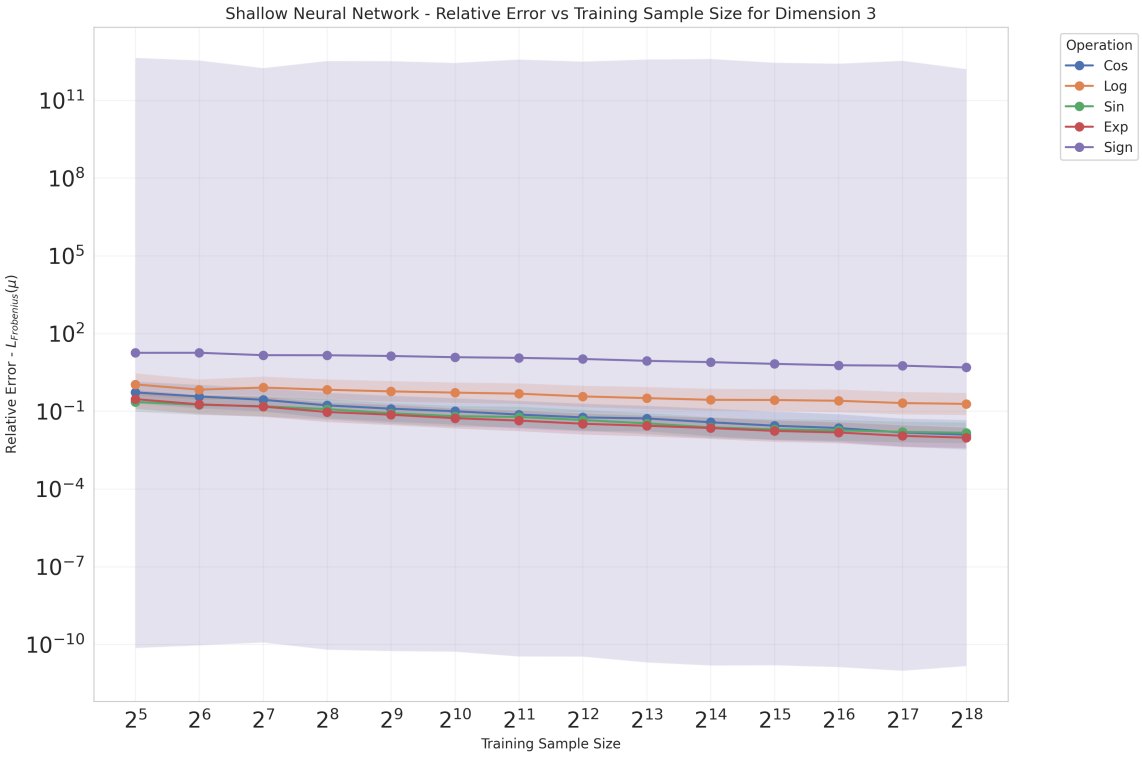


Figure 5.3: Shallow neural network results for 3×3 matrices

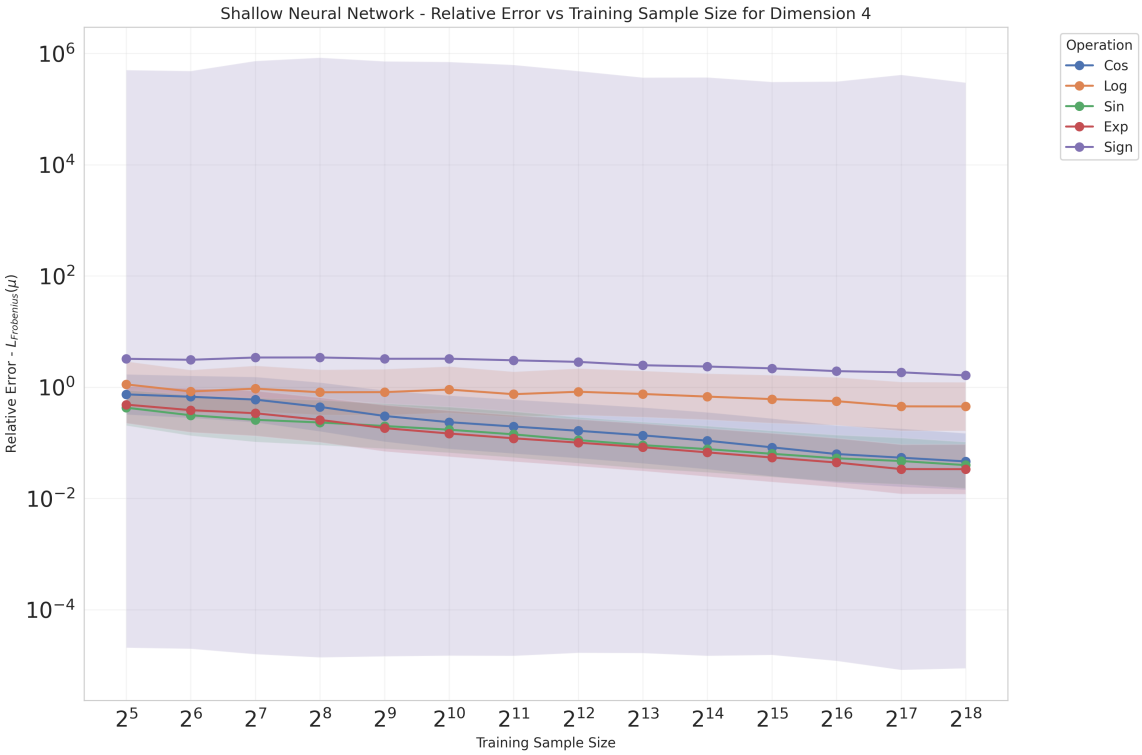


Figure 5.4: Shallow neural network results for 4×4 matrices

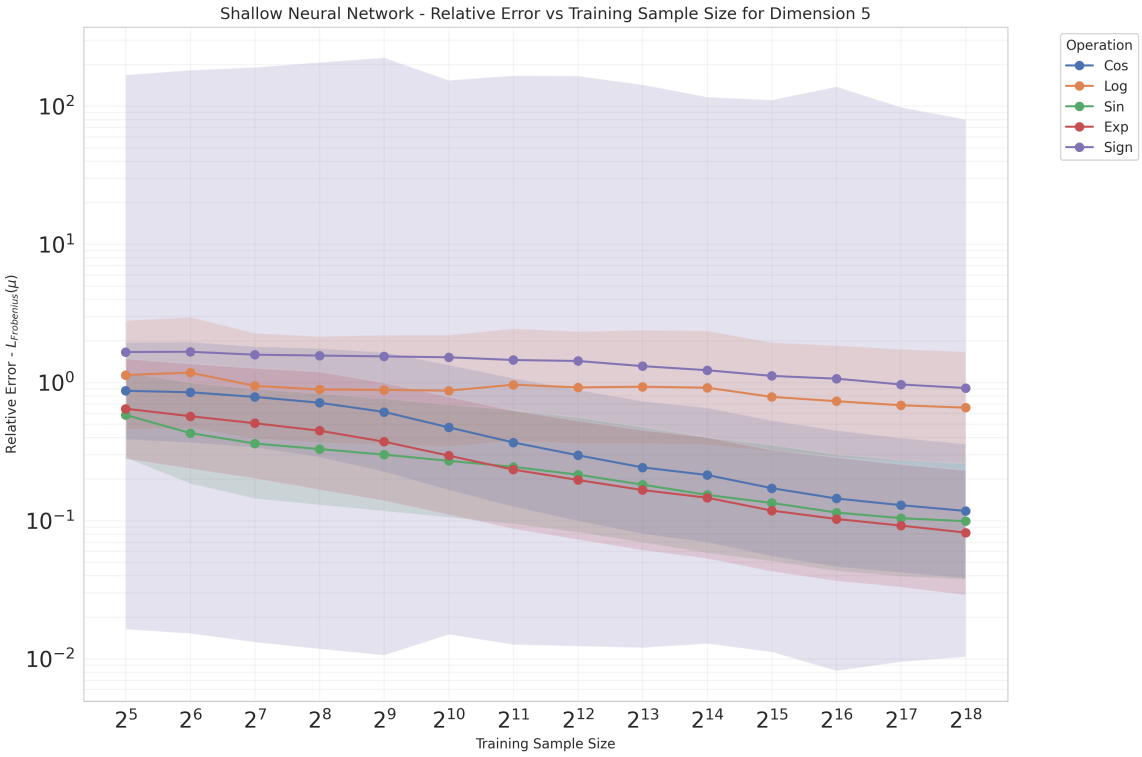


Figure 5.5: Shallow neural network results for 5×5 matrices

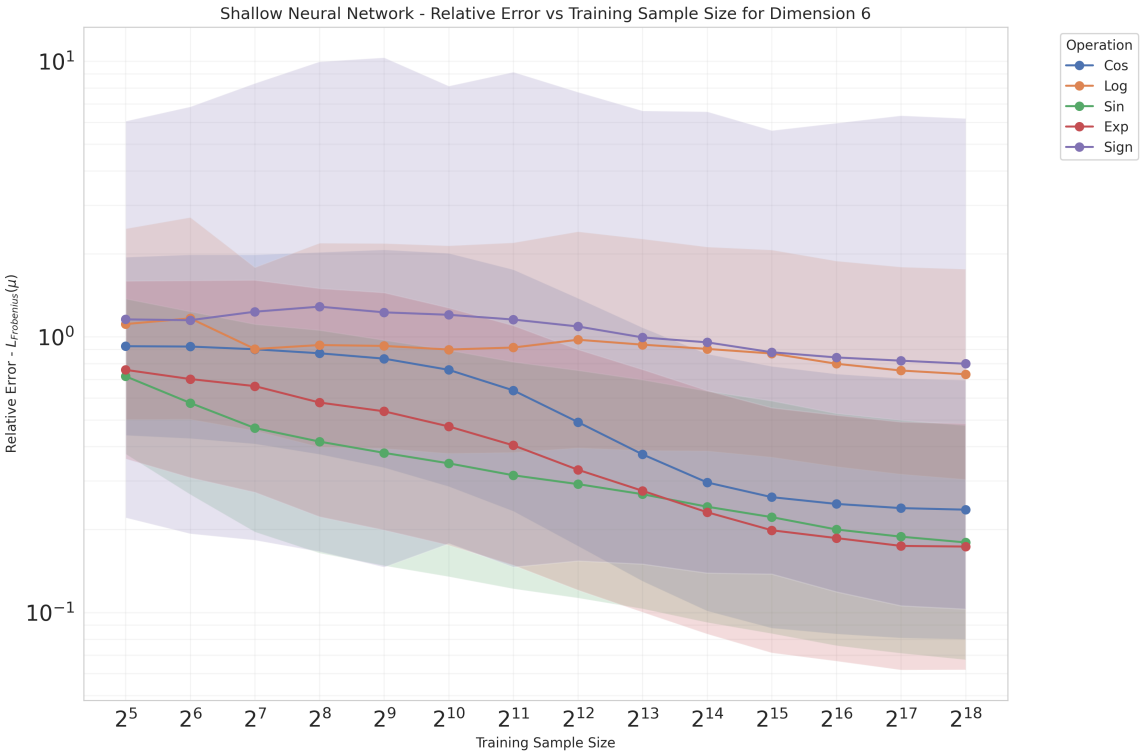


Figure 5.6: Shallow neural network results for 6×6 matrices

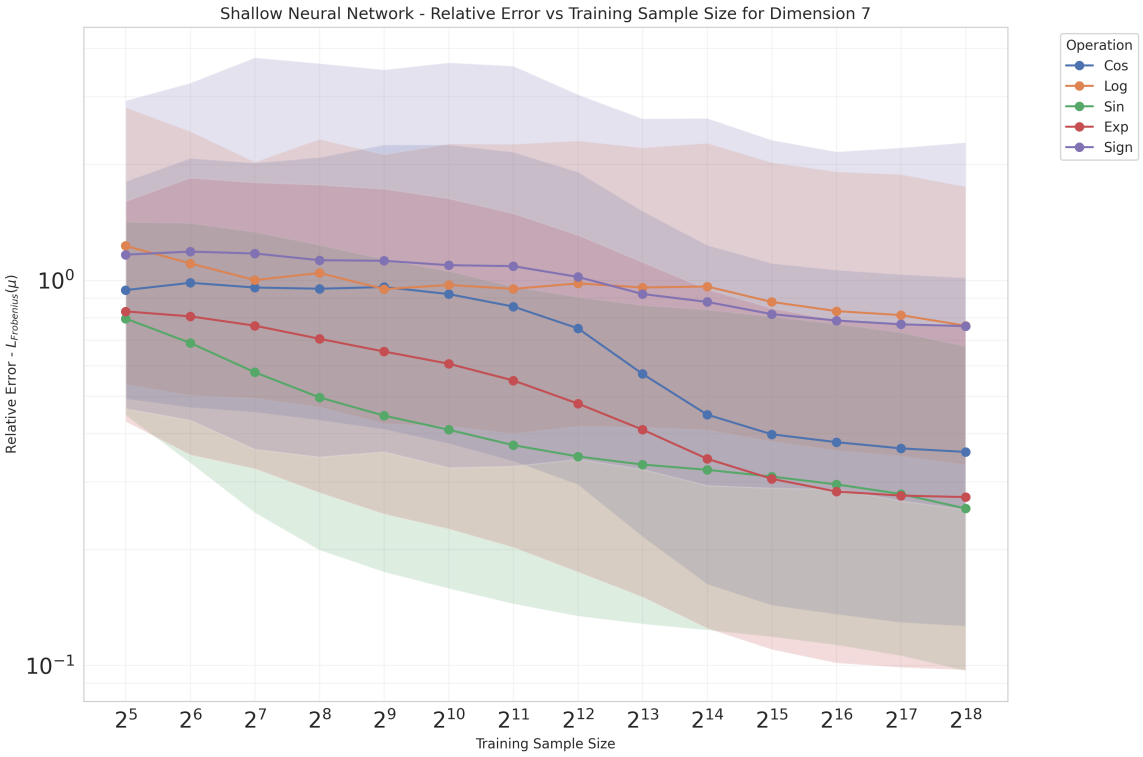


Figure 5.7: Shallow neural network results for 7×7 matrices

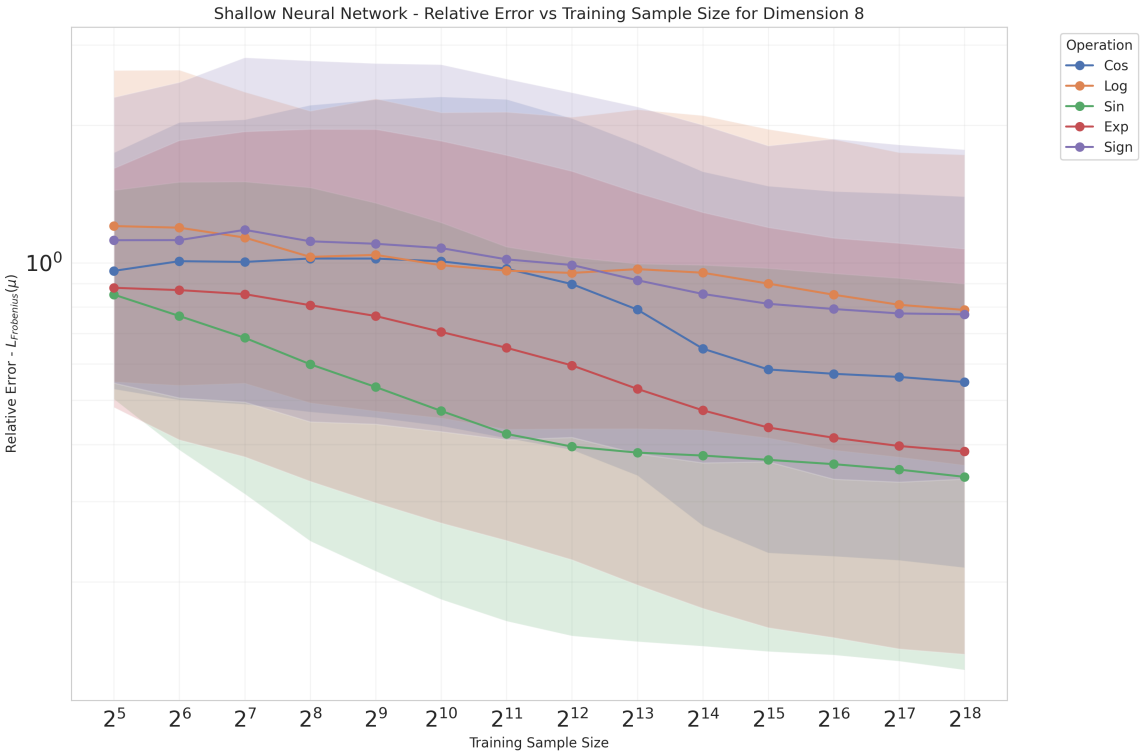


Figure 5.8: Shallow neural network results for 8×8 matrices

5.3.2 Deeper Neural Network

The following is the configuration for the deeper neural network:

- Input layer: Accepts flattened matrices of varying dimensions (1×1 to 8×8)
- Hidden layers: Seven layers with sizes 128, 256, 512, 1024, 512, 256, and 128 neurons respectively
- Output layer: Matches input dimension to reconstruct the transformed matrix
- Activation functions: ReLU for hidden layers, linear activation for output layer
- Optimizer: Adam[55] with learning rate 1×10^{-3}
- Batch size: 128
- Training epochs: 100
- Dropout rate: 0.2

5.3.2.1 Training and Testing Data

Similar to the shallow neural network in Section 5.3.1, we train $14 \times 5 \times 8 = 560$ models. 14 training set sizes ranging from 2^5 to 2^{18} samples and for each of the five matrix functions and each of the eight matrix dimensions. We use a test set of 2^{15} samples in all cases. The training and testing data is generated by sampling the matrix coefficients from a uniform distribution with coefficients in $[-1, 1]$.

5.3.2.2 Results

The results for the deeper neural network configuration are shown in Figures 5.9 through 5.16. These figures illustrate the performance across different matrix dimensions from 1×1 to 8×8 matrices. In general, this configuration performs worse than the shallow neural network configuration for all the matrix dimensions. This is likely due to the fact that the deeper neural network has more parameters to learn and it is more difficult to train for the same number of epochs [16, 97].

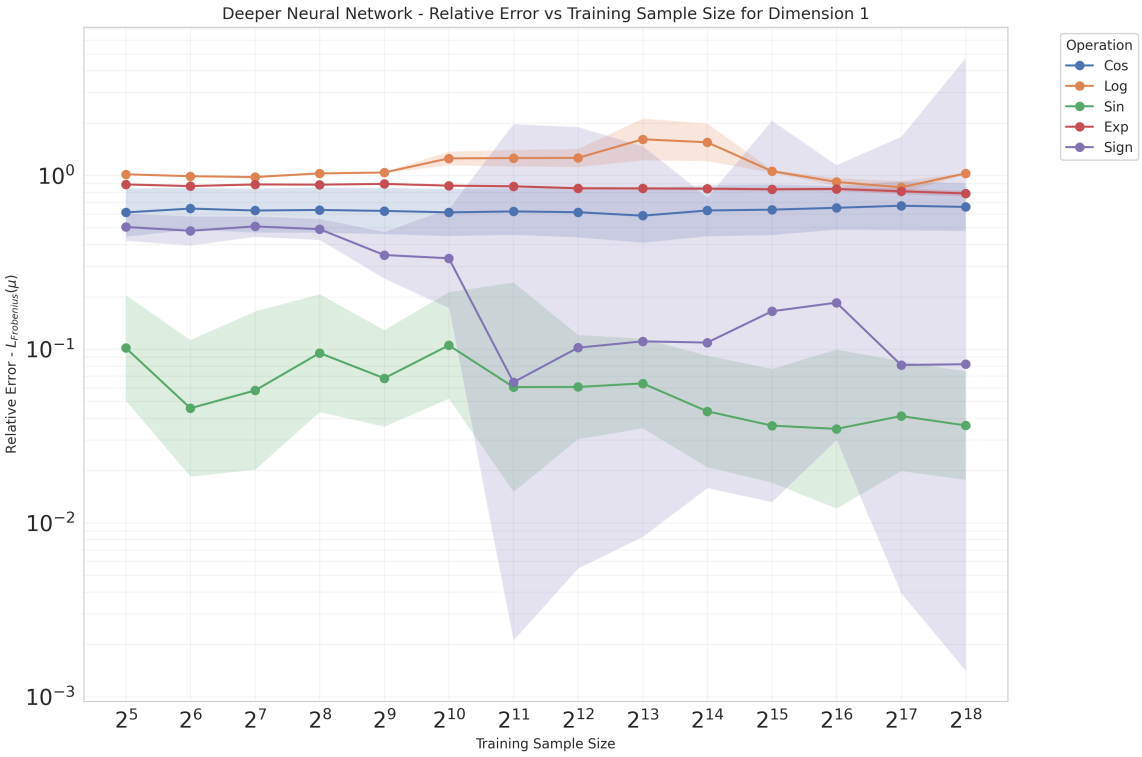


Figure 5.9: Deeper neural network results for 1×1 matrices

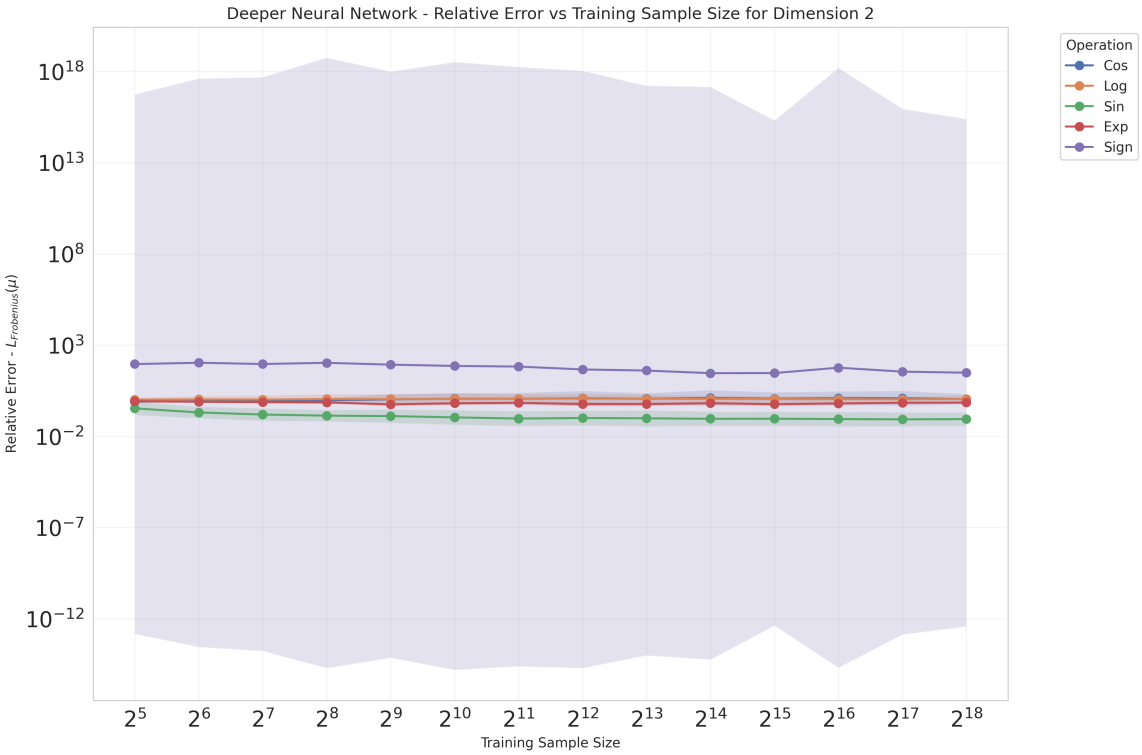


Figure 5.10: Deeper neural network results for 2×2 matrices

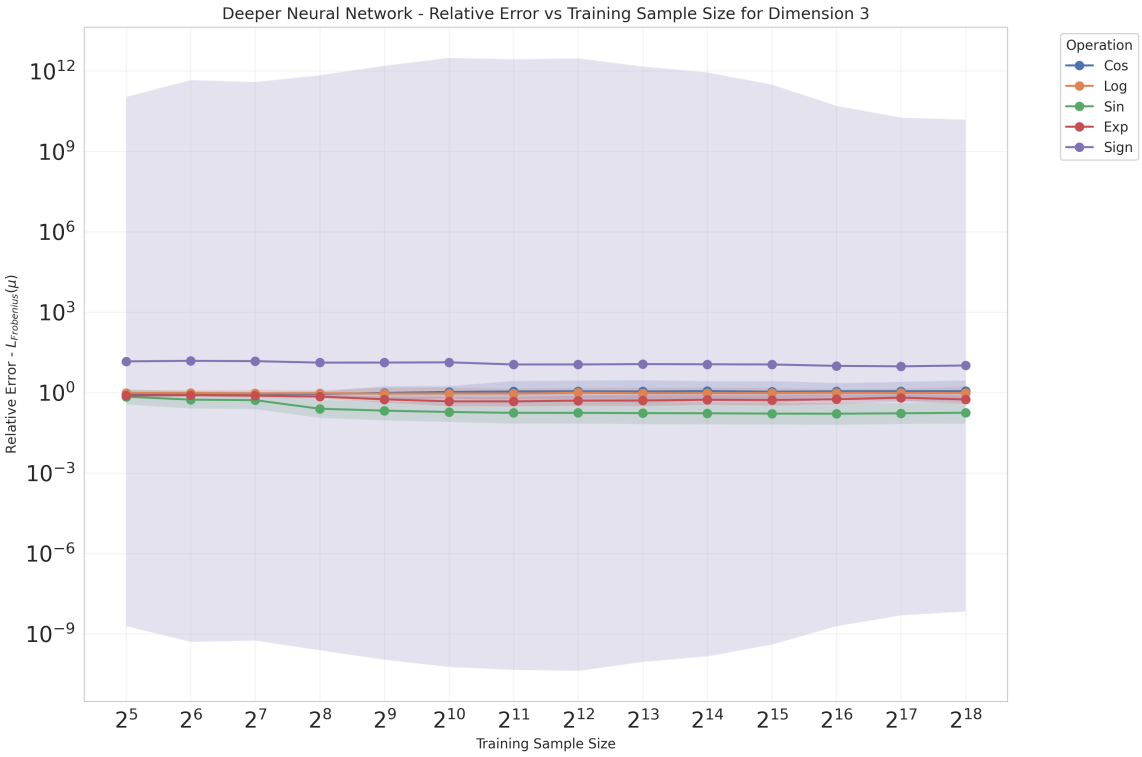


Figure 5.11: Deeper neural network results for 3×3 matrices

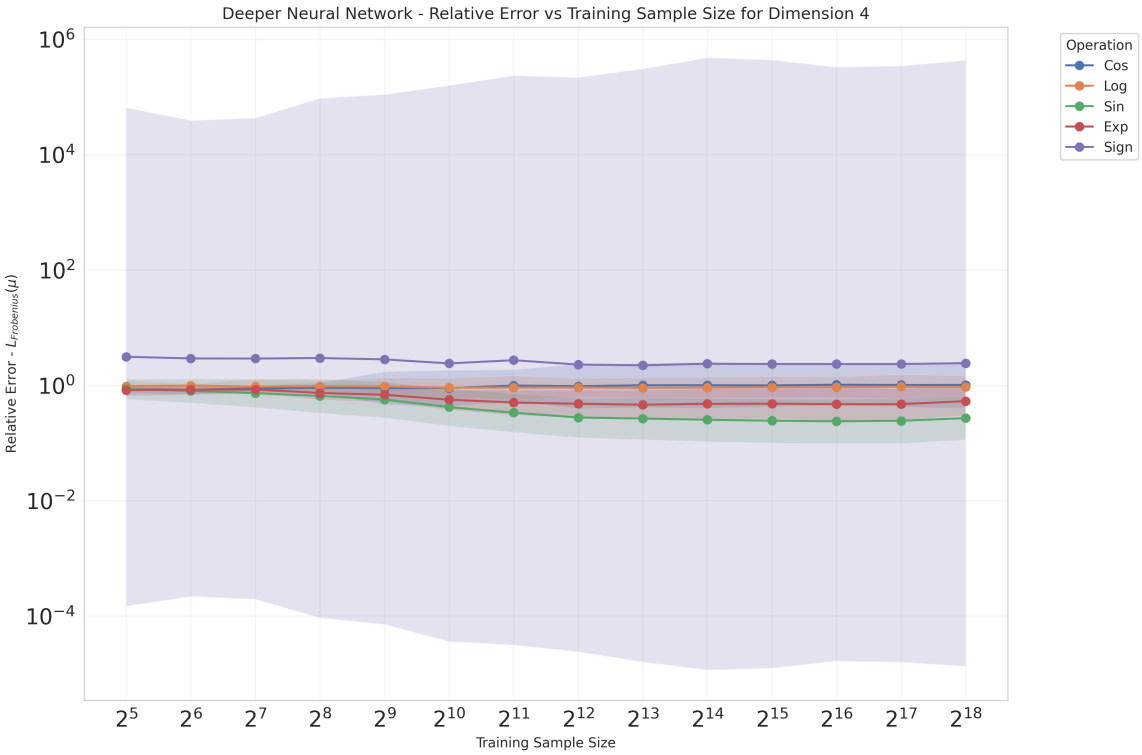


Figure 5.12: Deeper neural network results for 4×4 matrices

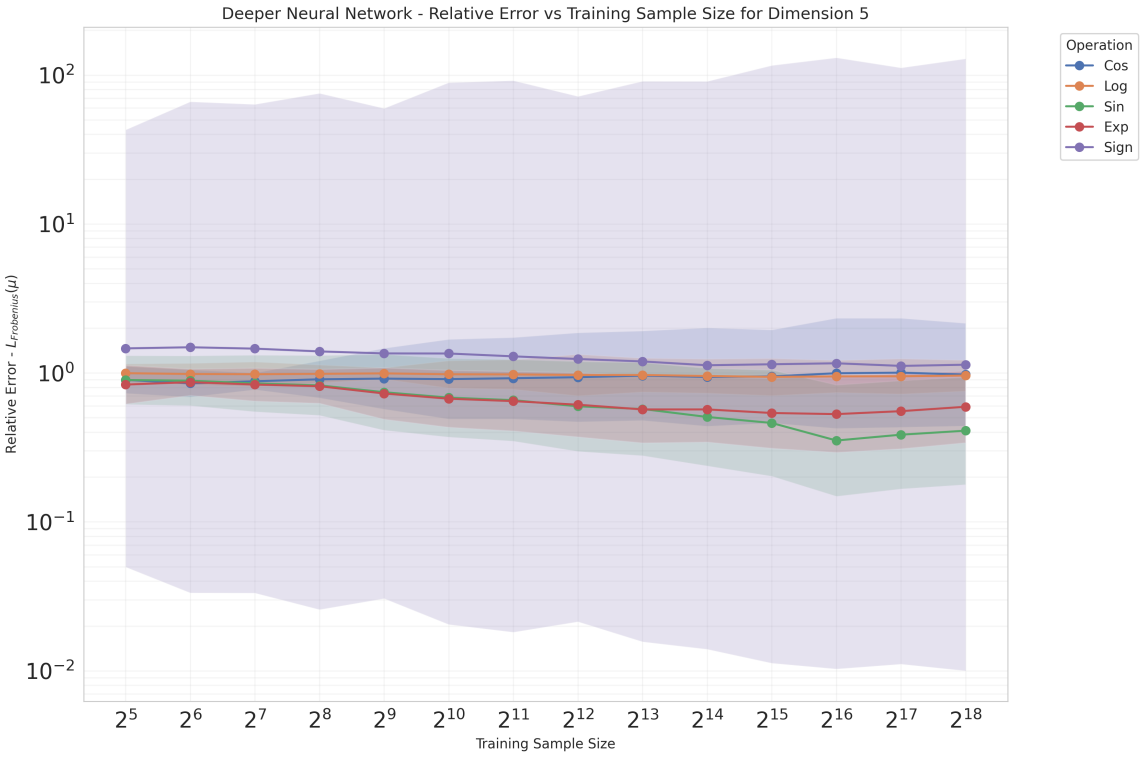


Figure 5.13: Deeper neural network results for 5×5 matrices

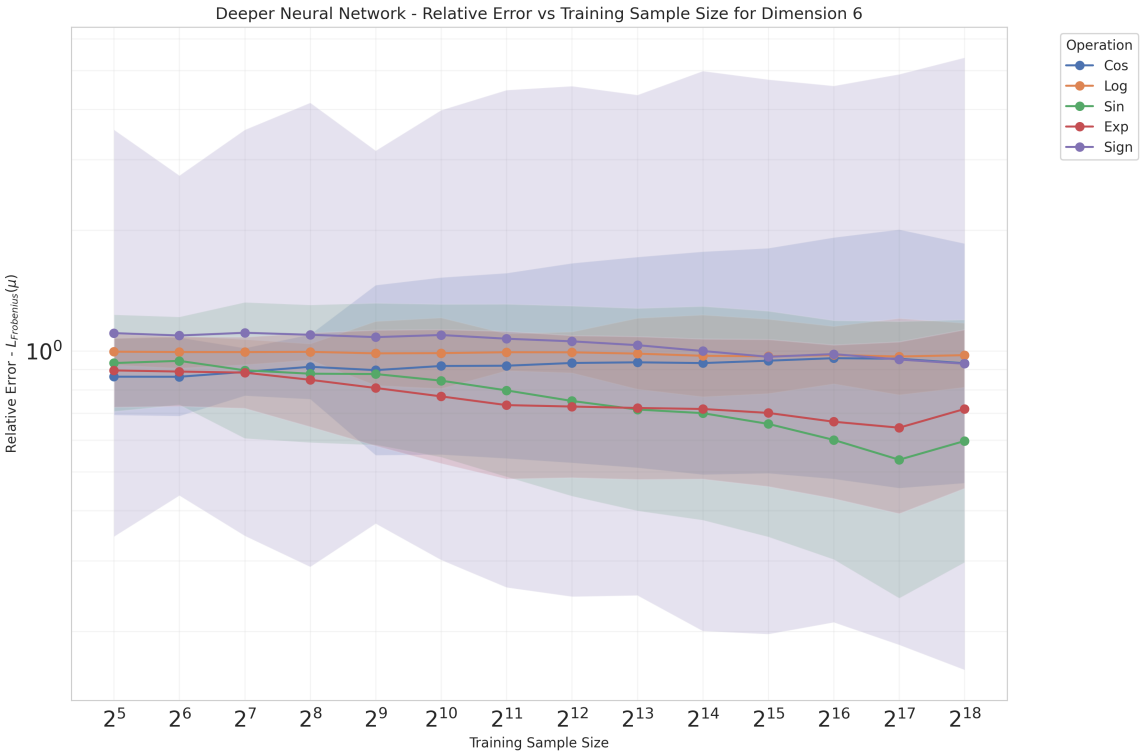


Figure 5.14: Deeper neural network results for 6×6 matrices

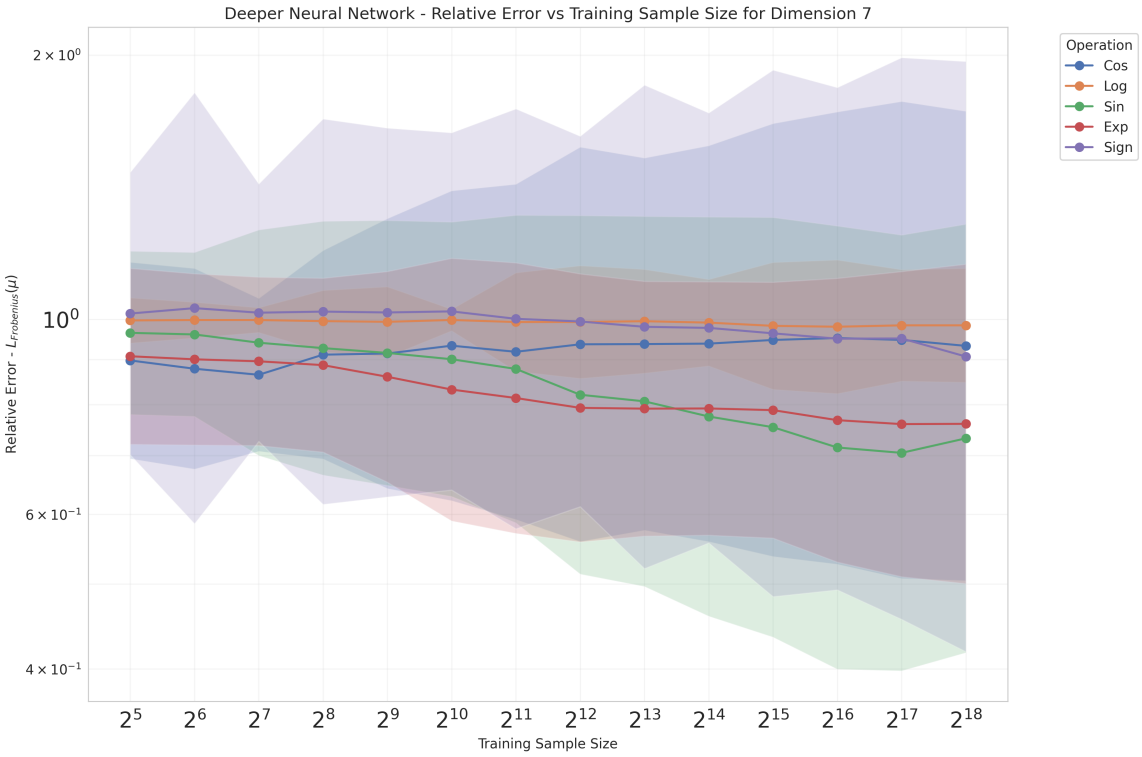


Figure 5.15: Deeper neural network results for 7×7 matrices

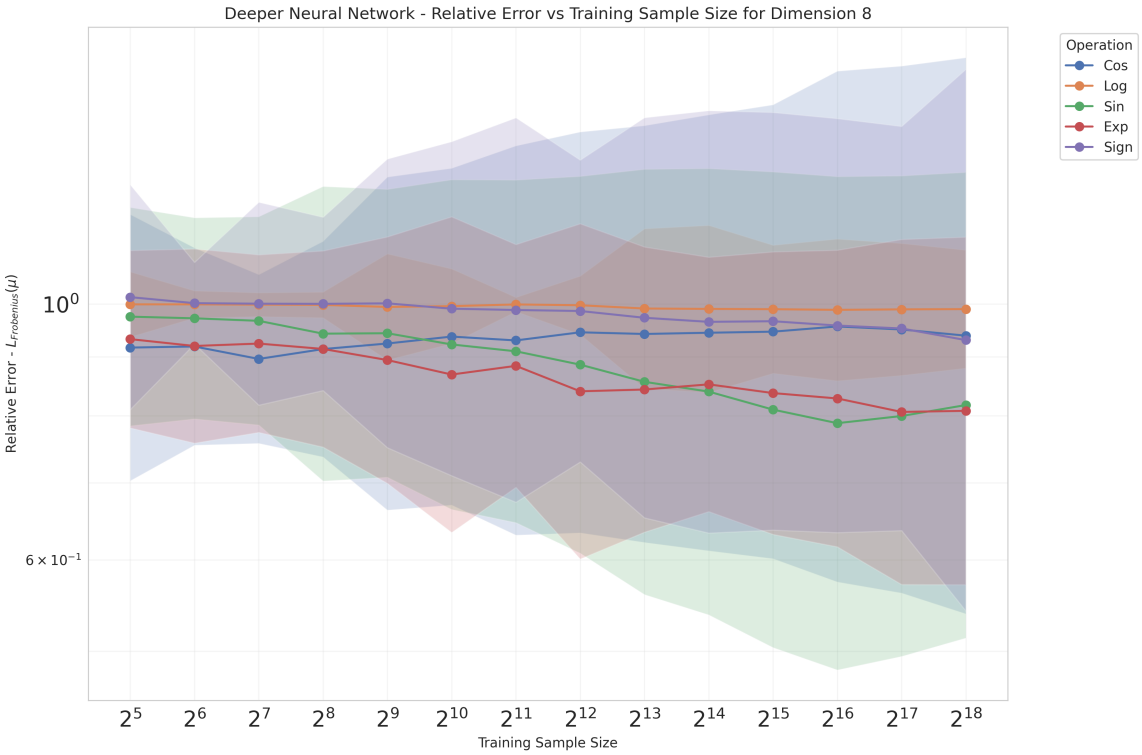


Figure 5.16: Deeper neural network results for 8×8 matrices

5.3.3 Transformer Encoder with Fourier Transform

In this experiment, we investigated the ability of a Transformer encoder architecture with Fourier feature encodings to approximate the matrix functions listed in Section 2.5. We conducted an initial experiment to determine if the matrix without encodings could be fed into the transformer encoder and to train it to approximate the matrix function. However, the results were not promising and it verified the results of Lee et al. (2023) which states that traditional training data is not optimal for the transformer in mathematical tasks. In order to overcome this issue, we will use the Fourier Transform to encode the matrix coefficients. Tancik et al. (2020) have shown that, using the Fourier transform in the case of a multi-layer perceptron (MLP) can be used to learn high frequency functions in low-dimensional domains (see [99]). We aim to see if this also holds for the transformer architecture. In this work, we will use the Fourier feature mappings [99] to attempt to improve the performance of the transformer architecture.

The Fourier feature mapping configuration enables learning of high-frequency functions in low-dimensional domains. The approach maps input $\mathbf{x}$ to a higher-dimensional space using:

$$\gamma(\mathbf{x}) = [\cos(2\pi\mathbf{B}\mathbf{x}), \sin(2\pi\mathbf{B}\mathbf{x})]$$

where $\mathbf{B}$ is a matrix with entries sampled from $\mathcal{N}(0, \sigma^2)$ and σ controls the frequency range of the embedding. This approach has been shown to provide several benefits including improved ability to represent high-frequency details and enables faster training with better generalization. In this experiment, we have tested large transformer models, the largest being a 16 layer transformer model with 64 heads. We have also increased the number of epochs to 600 (compared to 100 in the other experiments). This was done in order to ensure that the model has enough time to learn how to approximate the desired matrix function. We vary the number of transformer layers and the number of attention heads based on the matrix dimension d. This experiment was computationally heavy and took almost a month to train on a single Nvidia 3080 GPU. Due to the high computational cost, we have evaluated the model on matrices of dimension 3×3 and 5×5 only.

5.3.4 Transformer Encoder with Fourier Transform

The following is the configuration for the transformer encoder with Fourier Transform:

- Input dimension: d^2 where d is the matrix dimension
- Number of Fourier features: $3d^2$
- Model dimension: $6d^2$
- Number of attention heads: d^2
- Number of transformer layers: Variable (2, 4, 8 and 16 layers tested)
- Activation functions: ReLU [74] for feed-forward networks
- Optimizer: Adam [55] with learning rate 1×10^{-3}
- Batch size: 64
- Training epochs: 600

5.3.4.1 Training and Testing Data

The training and testing data is generated by sampling the matrix coefficients from a uniform distribution with coefficients in $[-1, 1]$. The three matrix dimensions that we have tested are 3×3, 5×5, and 8×8. We train $14 \times 5 \times 4 \times 3 = 840$ models. 14 training set sizes ranging from 2^5 to 2^{18} samples for each of the five matrix functions, for each of the four different transformer encoder configurations and for each of the three different matrix dimensions. We use a test set of 2^{15} samples.

5.3.4.2 Results for 3×3 matrices

The results of the experiments with the Fourier feature mappings using different numbers of transformer layers are shown in Figures 5.17, 5.18, 5.19, and 5.20. These figures demonstrate how the number of transformer layers affects the performance on 3×3 matrices. We observe poor performance in terms of relative error. The shaded plots show that there is not much success in the model learning the matrix function. The cosine function performs the best, but it is still not able to learn the function to a high degree of accuracy. The performance marginally improves with the number of transformer layers and the number of attention heads. While the performance is poor, we observe that the matrix cosine function performs better than on the deeper neural network however, it is still not able to learn the function to a high degree of accuracy. However, the shallow neural network still performs better on an average than the transformer encoder with Fourier Transform. We note that the matrix sign function is one of the worst performing matrix functions.

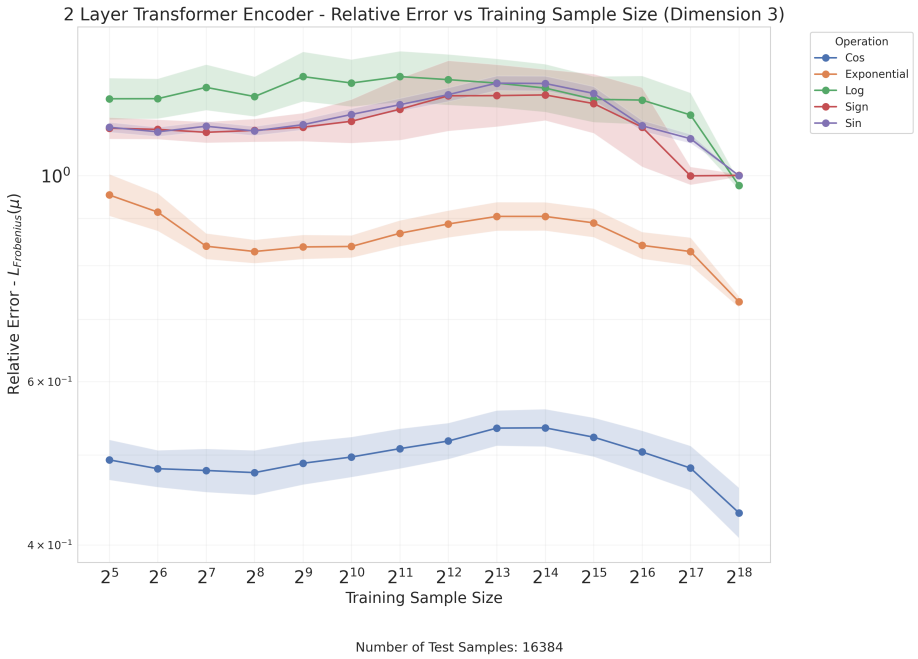


Figure 5.17: Performance of transformer with 2 layers using Fourier features on 3×3 matrices

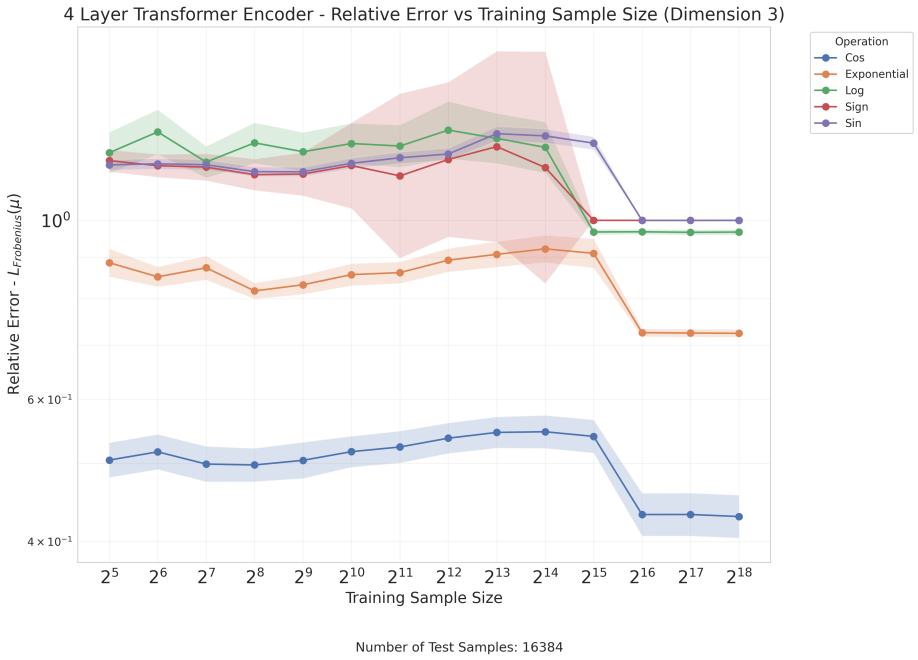


Figure 5.18: Performance of transformer with 4 layers using Fourier features on 3×3 matrices

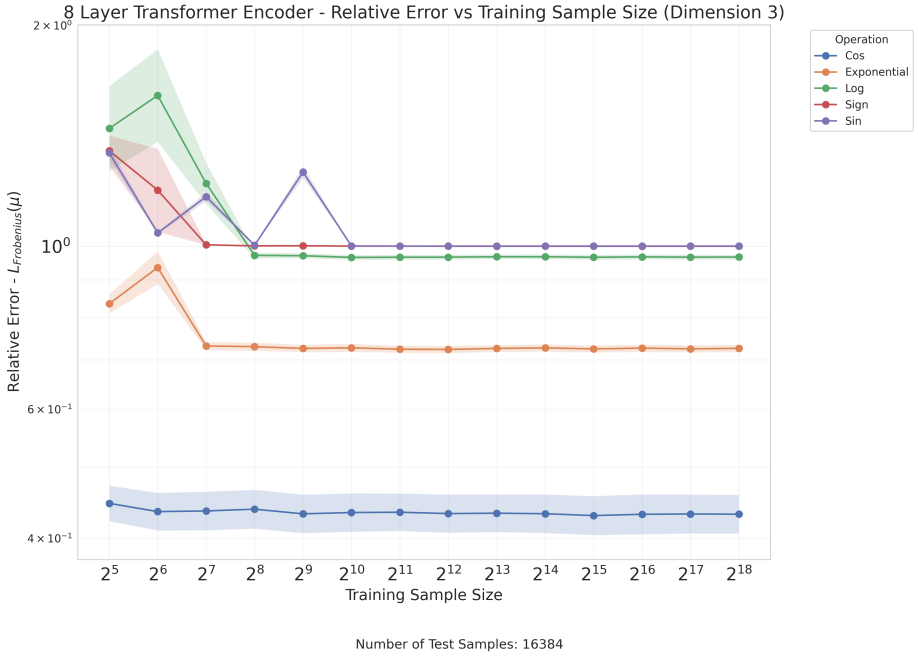


Figure 5.19: Performance of transformer with 8 layers using Fourier features on 3×3 matrices

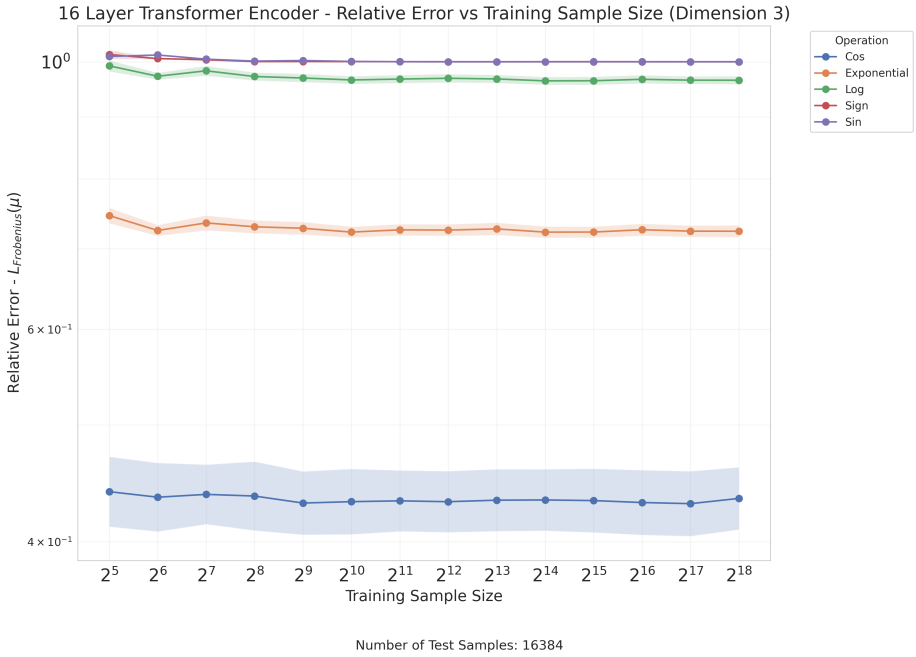


Figure 5.20: Performance of transformer with 16 layers using Fourier features on 3×3 matrices

5.3.4.3 Results for 5×5 matrices

The results of the experiments with the Fourier feature mappings using different numbers of transformer layers are shown in Figures 5.21, 5.22, 5.23, and 5.24. These figures demonstrate how the number of transformer layers affects the performance on 5×5 matrices. The performance is worse than the 3×3 matrices. We note that the matrix sign function is one of the worst performing matrix functions again.

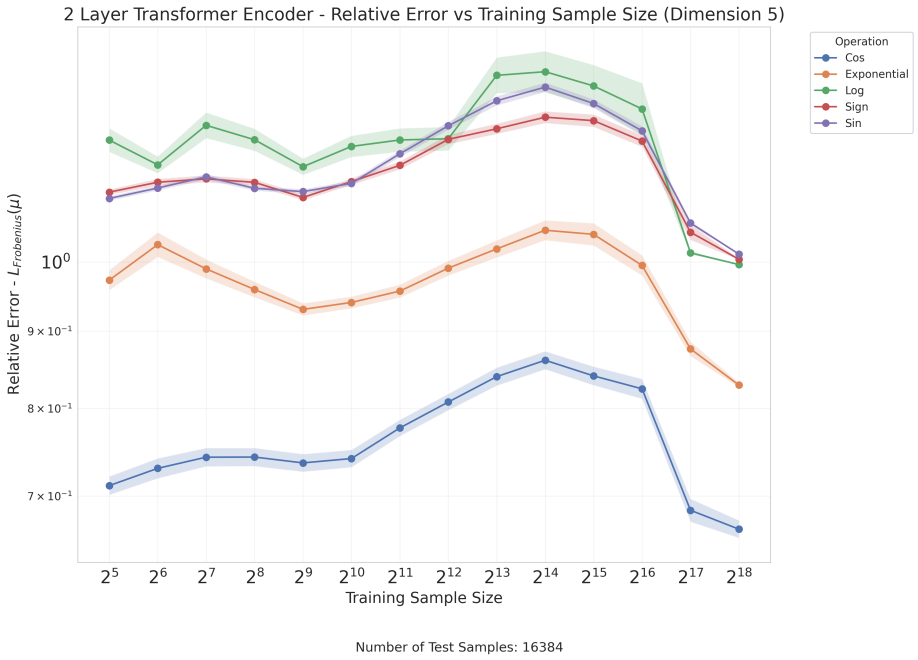


Figure 5.21: Performance of transformer with 2 layers using Fourier features on 5×5 matrices

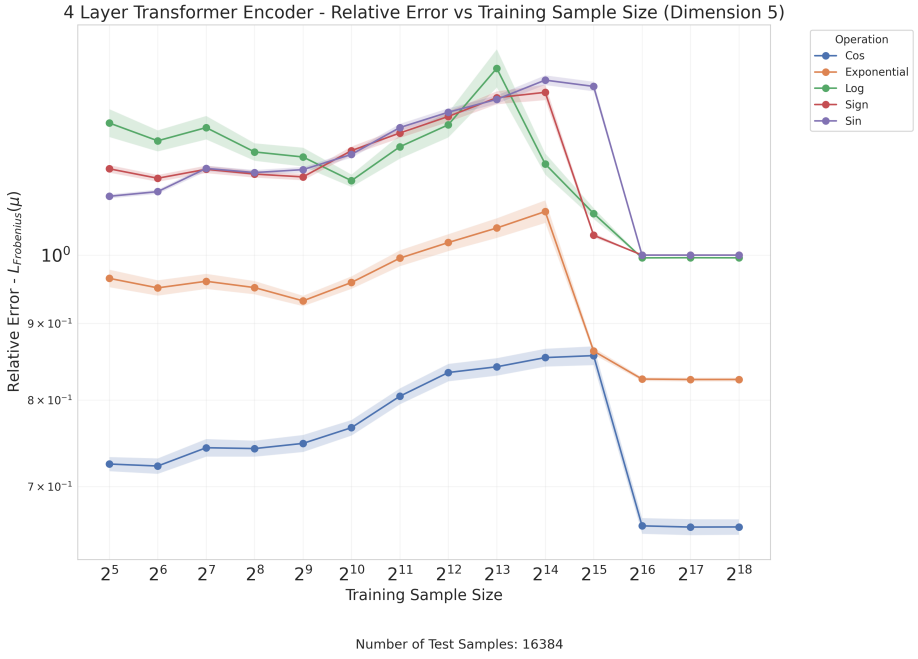


Figure 5.22: Performance of transformer with 4 layers using Fourier features on 5×5 matrices

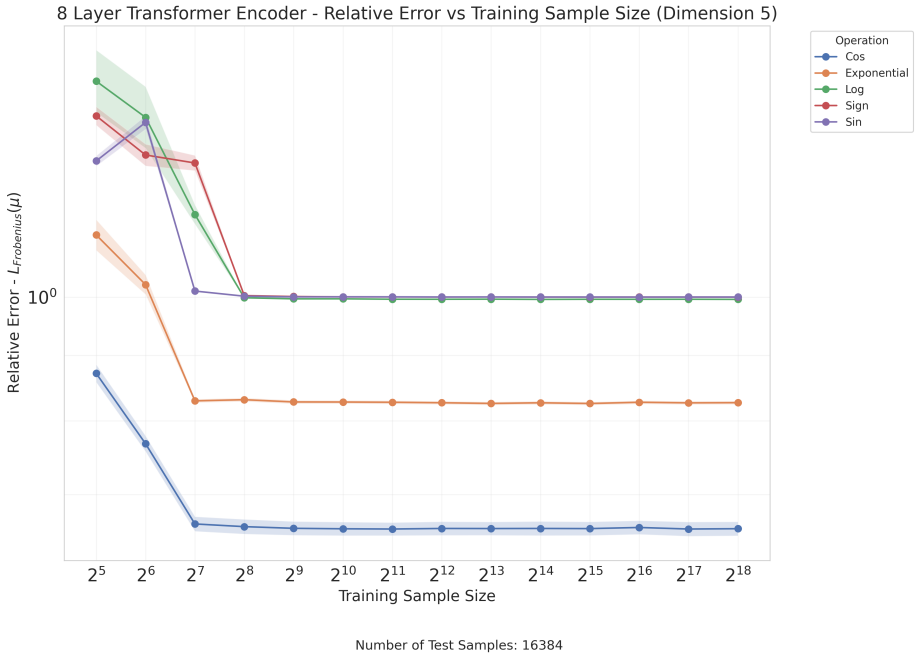


Figure 5.23: Performance of transformer with 8 layers using Fourier features on 5×5 matrices

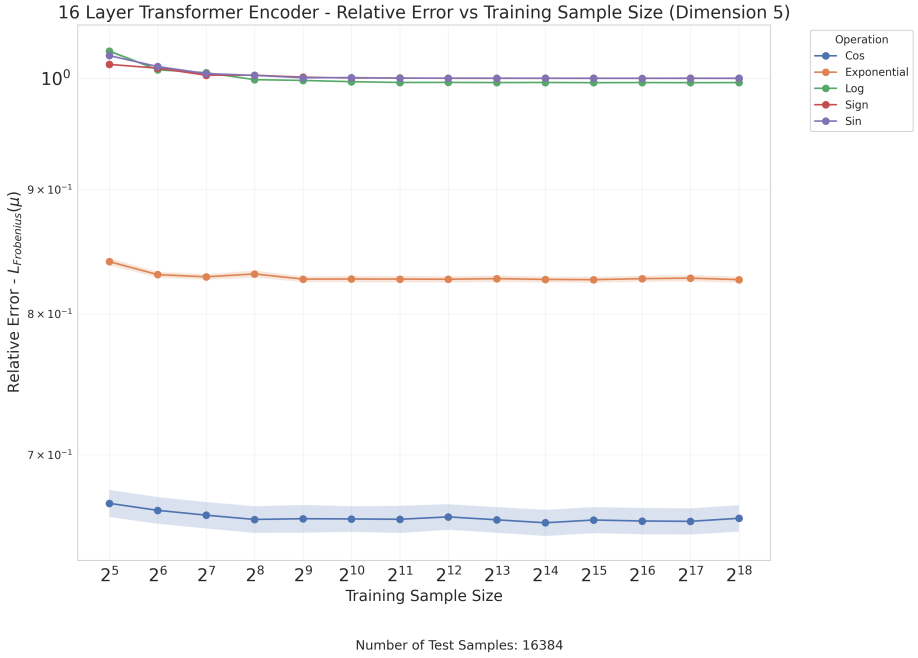


Figure 5.24: Performance of transformer with 16 layers using Fourier features on 5×5 matrices

5.3.4.4 Results for 8×8 matrices

The results of the experiments with the Fourier feature mappings using different numbers of transformer layers are shown in Figures 5.25, 5.26, 5.27, and 5.28. These figures demonstrate how the number of transformer layers affects the performance on 8×8 matrices. The performance is worse than the 3×3 and 5×5 matrices. The high dimensionality of the 8×8 matrices makes it difficult for the transformer encoder with Fourier Transform to learn the matrix function. We note that the matrix sign function is one of the worst performing matrix functions again. The model is unable to learn any of the matrix functions to a high degree of accuracy.

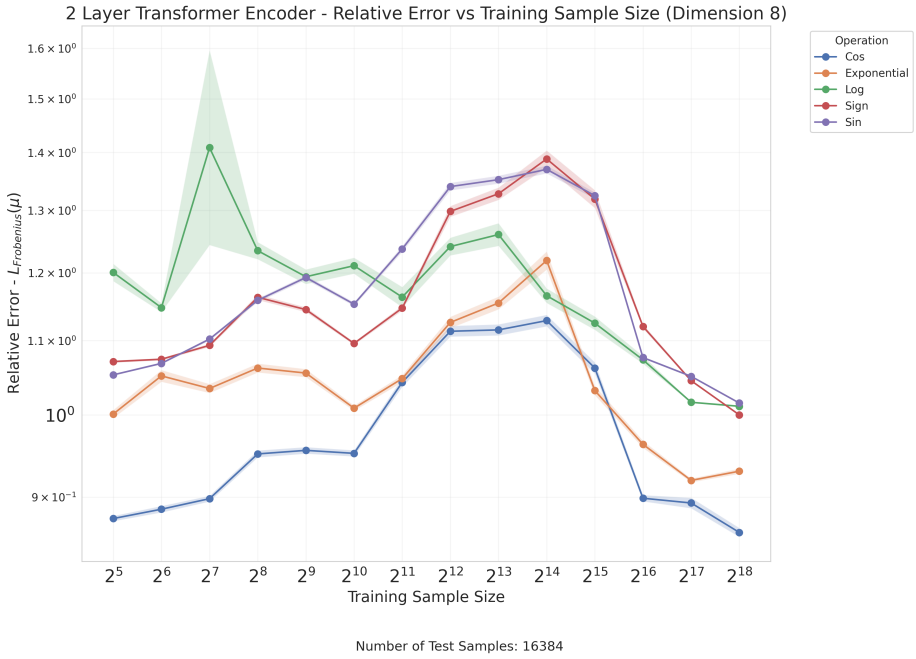


Figure 5.25: Performance of transformer with 2 layers using Fourier features on 8×8 matrices

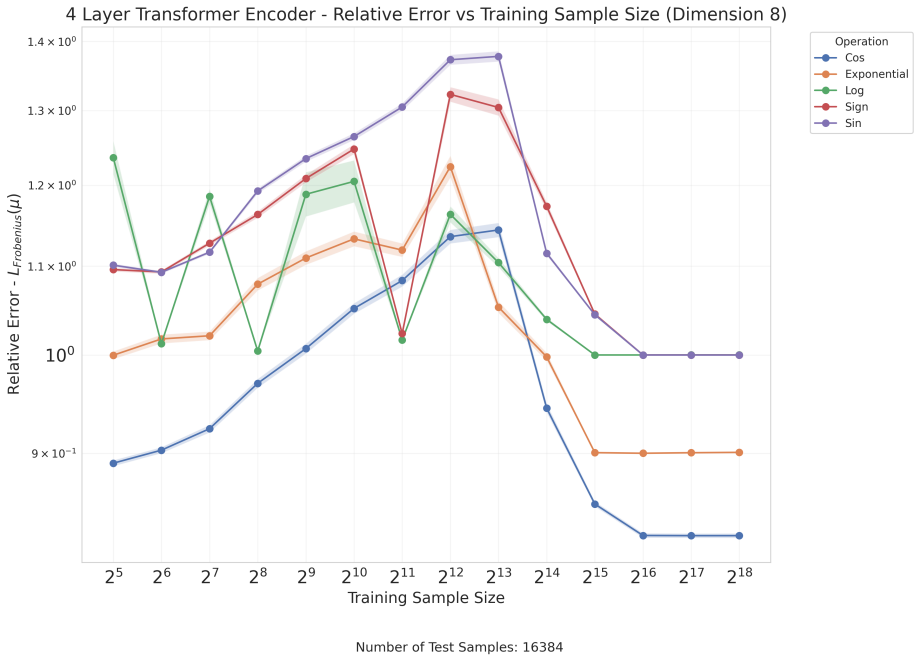


Figure 5.26: Performance of transformer with 4 layers using Fourier features on 8×8 matrices

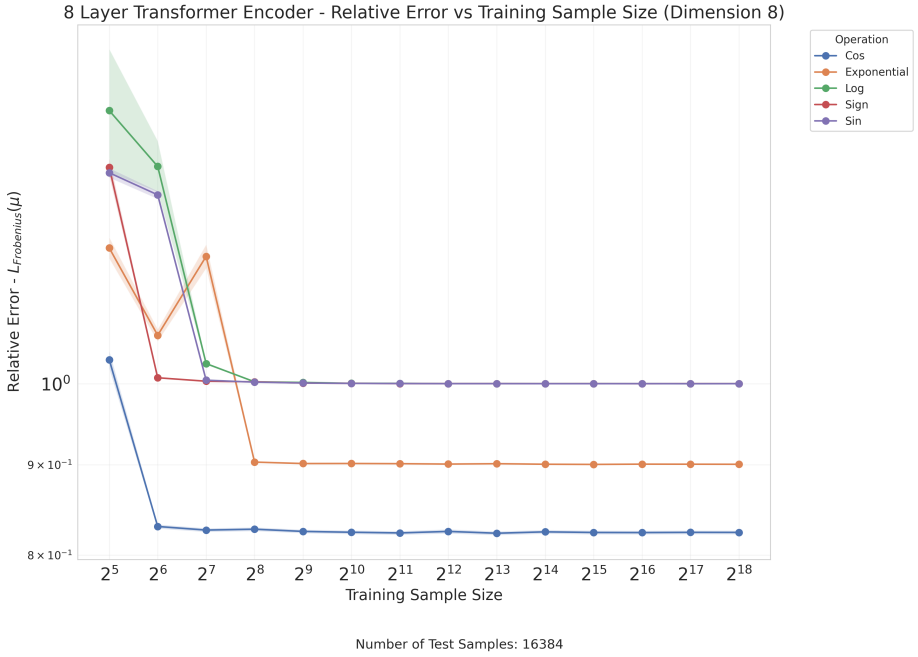


Figure 5.27: Performance of transformer with 8 layers using Fourier features on 8×8 matrices

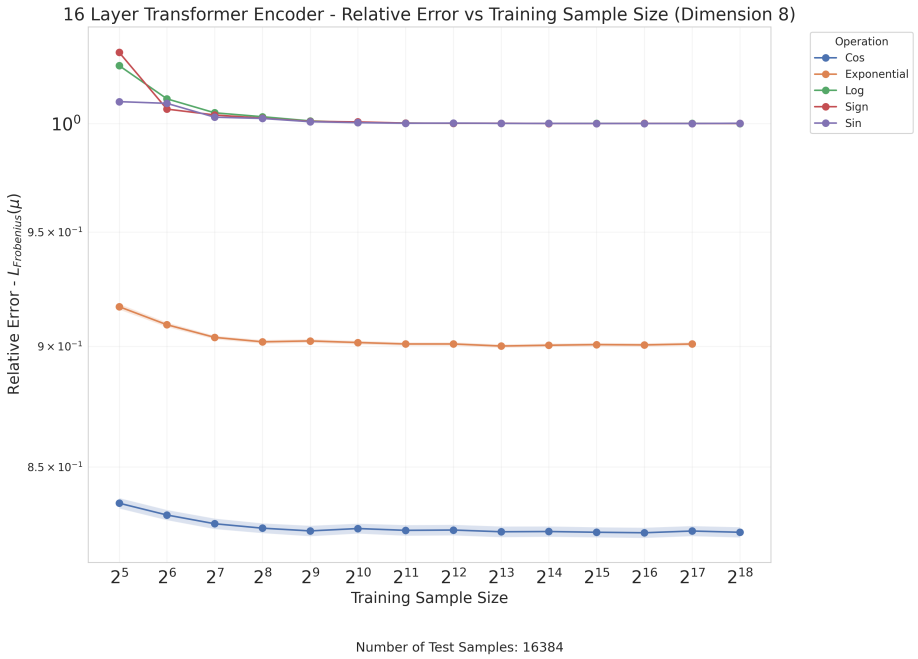


Figure 5.28: Performance of transformer with 16 layers using Fourier features on 8×8 matrices

5.3.5 Transformer Encoder-Decoder with Encodings

This section contains the results for the transformer encoder-decoder with encodings for 3×3 and 5×5 matrices. We use a transformer encoder-decoder with 8 layers of the encoder and 1 of the decoder with 8 heads of attention in each layer.

5.3.5.1 Encoding Scheme

The encoding scheme used affects the performance of the transformer architecture. Evidence of this is shown by Nogueira et al. (2021) (see [76]) where they have conducted experiments to show that the encoding scheme used in the transformer architecture significantly changes the answer to the question *"Can transformers add five digit numbers?"*. The results from this paper, show that the model fails to learn addition of five-digit numbers when using subwords (e.g., "32"), and it struggles to learn with character-level representations (e.g., "3 2"). By introducing position tokens (e.g., "3 10e1 2"), the model learns to accurately add and subtract numbers up to 60 digits. The primary paper used as a reference for this work, Charton (2021) (see [25]), has shown that different encoding schemes produce varied results in estimating the answer to certain problems in linear algebra using transformers.

In this work, we will use the encoding scheme proposed by Charton (2021) (see [25]) for the transformer architecture. This has proven to be effective for problems in linear algebra [25].

Any floating point number can be encoded using the following scheme [25]:

$$For \quad x \in \mathbb{R}, \quad x \approx s \cdot m \cdot 10^e, (s, m, e) \in \{-1, 1\} \times \{100, \dots, 999\} \times \mathbb{Z}$$

Based on this representation, we can encode the matrix coefficients into a sequence of tokens. We take four different encodings for the matrix coefficients as shown in Table 5.1.

Embedding	**3.14**	$\mathbf{-6.02 \times 10^{23}}$	**Tokens / coefficient**	**Vocabulary Size**
P10	[+, 3, 1, 4, E-2]	[−, 6, 0, 2, E21]	5	210
P1000	[+, 314, E-2]	[−, 602, E21]	3	1100
B1999	[314, E-2]	[−602, E21]	2	2000
FP15	[FP314/-2]	[FP-602/21]	1	30000

Table 5.1: Four encodings for matrix coefficients[25].

5.3.5.2 Training and Testing Data

All matrices have been generated using a gaussian distribution over the interval $[-5, 5]$. The matrices evaluated are of dimensions 3×3 and 5×5. We use the L1 norm (see Appendix D.2.1) to measure the error between the predicted and target matrices. Since attention to $\mathbb{R}^{n \times d}$ does not take ordering into consideration, we encode the position of a_{ij} in the embedding as Vi and Vj, which denote the encoded row and column index of $a_{i,j}$, respectively.

5.3.5.3 Configuration

The transformer architecture used in our experiments consists of an encoder with 8 layers and 8 attention heads, paired with a single-layer decoder also using 8 attention heads. This type of shallow decoder architecture was also used for experiments in Charton (2021) (see [25]) which yielded results with high accuracy. Both the encoder and decoder utilize an embedding dimension of 512. Shared input/output embeddings (based on Table 5.1) are used across the encoder and decoder. From a performance perspective, Automatic Mixed

Precision (AMP) dynamically adjusts the precision of computations, using lower-precision floating-point formats (e.g., fp16) where possible to improve performance while maintaining numerical stability. To improve training efficiency and memory utilization, we implemented mixed precision training (fp16) with AMP level 2 and applied gradient clipping [80] with a norm threshold of 5.0. As transformers are sensitive to the learning rate [67, 103], for training, we used the Adam [55] optimizer with a warmup schedule spanning 10,000 updates and a peak learning rate of 0.0001. The model was trained for 100 epochs, with each epoch processing 300,000 samples. We used a training batch size of 64 and an evaluation batch size of 128.

Our experimental setup focused on matrices of dimensions 3×3 and 5×5, with coefficients generated from a Gaussian distribution bounded within $[-5, 5]$. The evaluation dataset comprised 10,000 samples. For assessing model accuracy, we primarily used a float tolerance of 0.05 (5%), with additional evaluations at stricter tolerances of 0.02 (2%), 0.01 (1%) and 0.005 (0.5%) to gauge precision. The training infrastructure utilized a single GPU setup with 10 worker processes dedicated to data loading, ensuring efficient throughput during training.
We forked the code from the GitHub repository github.com/facebookresearch/LAWT and modified it to suit our needs. Our code was written in PyTorch [81] and the code can be found in the GitHub repository github.com/rahul3/LAWT.

5.3.5.4 Results for 3×3 matrices

For 3×3 matrices, we conducted a comprehensive evaluation across different matrix functions and encoding schemes, with results shown in Table 5.2.

Operation	**Encoding**	**Accuracy**			
		tol=0.05	**tol=0.02**	**tol=0.01**	**tol=0.005**
Exponential	P10	93.63%	34.42%	4.15%	0.12%
Logarithm	P10	91.88%	84.82%	71.72%	46.19%
Sign	P10	96.73%	91.87%	76.90%	44.29%
Sine	P10	40.92%	11.21%	0.57%	0.00%
Cosine	P10	10.32%	2.24%	0.05%	0.00%
Exponential	P1000	87.58%	37.95%	10.30%	1.17%
Logarithm	P1000	92.89%	86.62%	74.51%	46.37%
Sign	P1000	**97.52%**	**95.24%**	**88.21%**	**65.57%**
Sine	P1000	8.38%	5.94%	1.57%	0.16%
Cosine	P1000	10.64%	4.62%	0.43%	0.01%
Exponential	FP15	0.00%	0.00%	0.00%	0.00%
Logarithm	FP15	0.00%	0.00%	0.00%	0.00%
Sign	FP15	**96.33%**	**92.42%**	**86.90%**	**66.72%**
Sine	FP15	0.00%	0.00%	0.00%	0.00%
Cosine	FP15	0.00%	0.00%	0.00%	0.00%
Exponential	B1999	98.52%	74.73%	28.14%	3.11%
Logarithm	B1999	92.62%	79.96%	56.95%	22.05%
Sign	B1999	93.77%	85.45%	62.39%	29.18%
Sine	B1999	0.00%	0.00%	0.00%	0.00%
Cosine	B1999	0.04%	0.00%	0.00%	0.00%

Table 5.2: Accuracy results for different matrix functions and encodings on 3×3 matrices across various error tolerances.

The results demonstrate varying performance across different encoding schemes and matrix functions. The results in Table 5.2 highlight the varying effectiveness of different encoding schemes for approximating matrix functions using transformers. The sign function exhibits the highest overall accuracy across all encodings, The sign function exhibits the highest overall accuracy across all encodings, with the P1000 and FP15 encodings achieving over 92% accuracy at tol = 0.02, and maintaining strong performance at stricter tolerances, notably over 86% at tol = 0.01. In contrast, the exponential function's accuracy drops sharply as tolerance tightens, with P10, P1000 and B1999 encodings performing better than FP15. The B1999 encoding performs better than P10 and P1000 at tol = 0.02. The FP15 encoding fails completely for the exponential, logarithm, sine, and cosine functions, indicating poor numerical stability in this format. The B1999 encoding, while effective for exponential at tol = 0.05, degrades significantly for stricter tolerances. sine and cosine functions generally perform poorly, with accuracies close to zero. These results validate the importance of choosing an appropriate encoding scheme based on the target function and desired error tolerance. The most significant result is that the sign function performs the best across all encodings and error tolerances and the transformer encoder-decoder with encodings is able to learn the function to an accuracy of 88.21% at tol = 0.01.

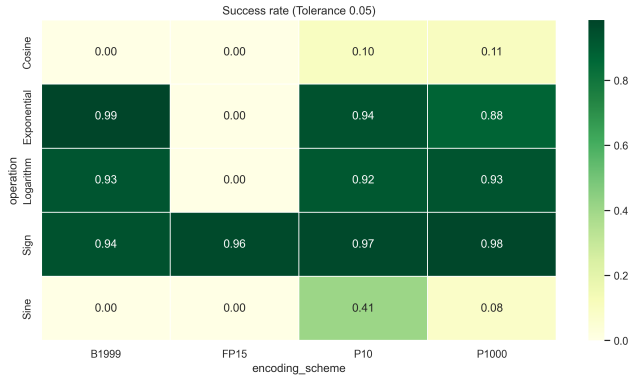


Figure 5.29: Performance at tolerance 0.05 on 3×3 matrices

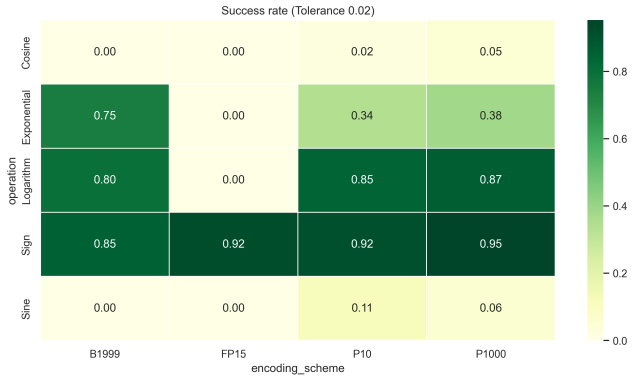


Figure 5.30: Performance at tolerance 0.02 on 3×3 matrices

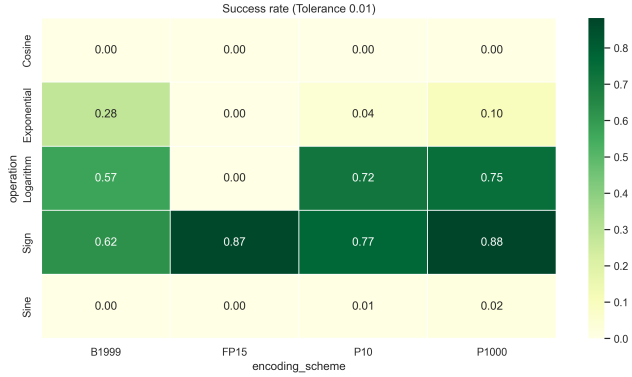


Figure 5.31: Performance at tolerance 0.01 on 3×3 matrices

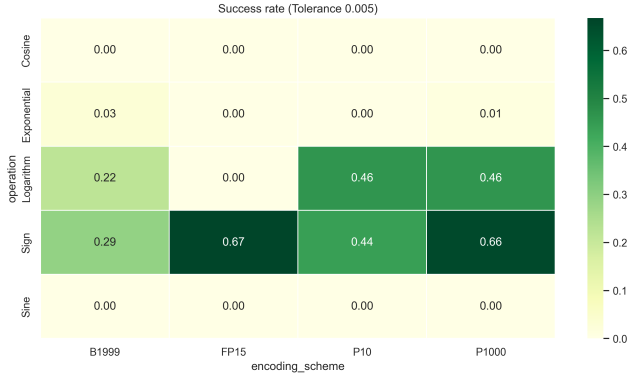


Figure 5.32: Performance at tolerance 0.005 on 3×3 matrices

5.3.5.5 Results for 5×5 matrices

The results in Table 5.3 demonstrate a significant drop in accuracy compared to the 3×3 case, highlighting the increased difficulty in approximating matrix functions as dimensionality grows. The P10 and FP15 encodings fail completely across all operations, suggesting that they are inadequate for higher-dimensional matrices. P1000 achieves moderate performance for the exponential and logarithm functions at loose tolerances but rapidly degrades as stricter thresholds are applied. The B1999 encoding emerges as the best-performing scheme, particularly for sign, where it maintains an impressive 97.77% accuracy at tol $= 0.05$ and retains reasonable performance even as tolerance tightens. However, B1999 completely fails for logarithm, sine, and cosine, indicating its limited generalizability. Across all encodings, the matrix functions sine and cosine remain the most challenging to approximate, with accuracies consistently at zero. Our key result is that the transformer encoder-decoder with encodings is able to learn the sign function to a high degree of accuracy (over 82%) at tol $= 0.01$ and the performance improves at higher tolerances.

Operation	**Encoding**	**Accuracy**			
		tol=0.05	**tol=0.02**	**tol=0.01**	**tol=0.005**
Exponential	P10	0.00%	0.00%	0.00%	0.00%
Logarithm	P10	0.00%	0.00%	0.00%	0.00%
Sign	P10	0.00%	0.00%	0.00%	0.00%
Sine	P10	0.00%	0.00%	0.00%	0.00%
Cosine	P10	0.00%	0.00%	0.00%	0.00%
Exponential	P1000	85.97%	21.13%	1.18%	0.00%
Logarithm	P1000	88.44%	77.93%	59.20%	19.30%
Sign	P1000	89.99%	57.62%	11.38%	2.16%
Sine	P1000	0.00%	0.00%	0.00%	0.00%
Cosine	P1000	0.00%	0.00%	0.00%	0.00%
Exponential	FP15	0.00%	0.00%	0.00%	0.00%
Logarithm	FP15	0.00%	0.00%	0.00%	0.00%
Sign	FP15	1.26%	1.26%	1.26%	1.26%
Sine	FP15	0.00%	0.00%	0.00%	0.00%
Cosine	FP15	0.00%	0.00%	0.00%	0.00%
Exponential	B1999	95.95%	42.99%	14.14%	2.49%
Logarithm	B1999	0.00%	0.00%	0.00%	0.00%
Sign	B1999	**97.77%**	**95.13%**	**82.31%**	**32.04%**
Sine	B1999	0.00%	0.00%	0.00%	0.00%
Cosine	B1999	0.00%	0.00%	0.00%	0.00%

Table 5.3: Accuracy results for different matrix functions and encodings on 5×5 matrices across various error tolerances.

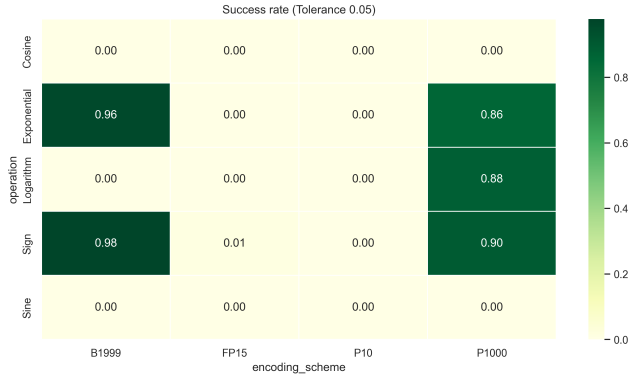


Figure 5.33: Performance at tolerance 0.05 on 5×5 matrices

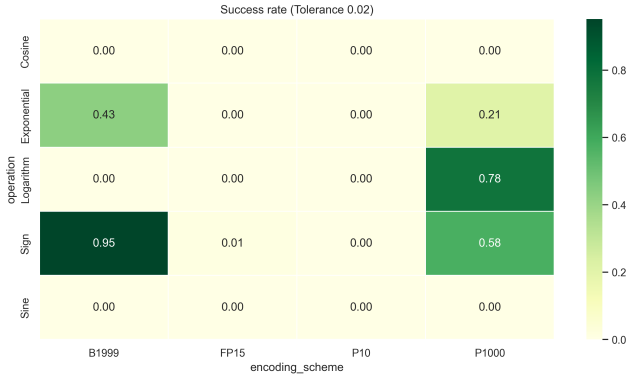


Figure 5.34: Performance at tolerance 0.02 on 5×5 matrices

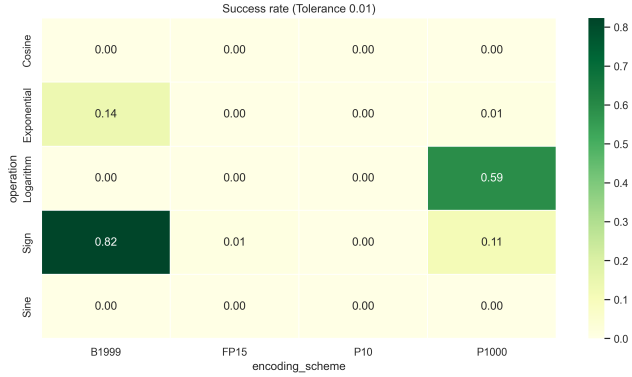


Figure 5.35: Performance at tolerance 0.01 on 5×5 matrices

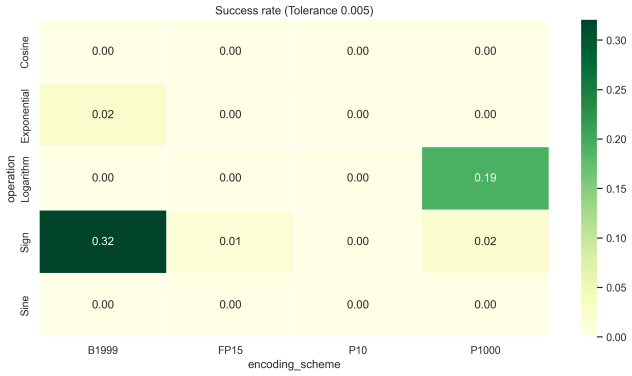


Figure 5.36: Performance at tolerance 0.005 on 5×5 matrices

Chapter 6

Conclusions and Future Work

This section contains the conclusions of the thesis and the future work that can be done to improve the results.

6.1 Conclusions

We have two main results. Our first main result is that we have proved a theorem (Theorem 4.3.1) bounding the width and depth of a ReLU [74] DNN in the approximation of the matrix exponential function. Our second main result is that we have shown that a transformer with 8 layers of the encoder and 1 of the decoder with 8 heads of attention in each layer can approximate the matrix sign function, at an accuracy of 88.21% for 3×3 matrices (using the FP15 encoding from Table 5.1) and an accuracy of 82.31% for 5×5 matrices (using the B1999 encoding from Table 5.1) at a tolerance of 1% with two significant digits. This result may be of interest to areas of research which have shown that transformer models can discover new Lyapunov functions [6] as the matrix sign function is used to solve Lyapunov equations, which, in turn, help find Lyapunov functions.

We have also shown that the encoding scheme in conjunction with the architecture used is highly important. The transformer encoder is not able to learn the matrix functions we are interested in using Fourier encodings however, the transformer encoder-decoder model with different encodings is able to learn some of them to a degree of accuracy.

6.2 Limitations and Future Work

There are several limitations to our approach. In Theorem 4.3.1, the sharpness of the bounds can be improved. Our proof strategy can probably be optimized and this could lead to improved width and depth bounds. We could improve the choice of K. Another improvement could be the construction of a network to approximate the matrix power A^k (needed for the Taylor expansion) done in a recursive manner. In future work, it would also be interesting to extend the theorem to consider other matrix functions. An analogous theorem for transformers is also an open problem.See Section 4.4 for further details.

In the training data used for the experiments of the shallow neural network (Section 5.3.1), the deep neural network (Section 5.3.2) and the transformer encoder with Fourier encoding (Section 5.3.4), we used matrices with coefficients in $[-1, 1]$ sampled from a uniform distribution. In the main experiment however, we used matrices with coefficients in

$[-5, 5]$ sampled from a normal distribution. We aim to extend our experiments to include matrices with coefficients in $[-5, 5]$ sampled from a normal distribution and also conduct experiments with training and test data generated from other distributions to see if the results are consistent. The transformer encoder-decoder model is for coefficients with only two significant digits. It would be interesting to extend the experiments to find a way to generalize the transformer encoder-decoder model to include matrices with coefficients with more significant digits which would help in accurate surrogate modelling of the matrix functions.

The transformer architecture used in our main experiment (in Section 5.3.5) is not able to be used for the other matrix functions. There is a need to find a more general architecture that can be used for the other matrix functions. Our experiments are also limited to certain matrix sizes upto 8×8 as the computational complexity of the matrix functions increases rapidly with the size of the matrix. A more general architecture is needed to be able to handle larger matrices. We have also used only a single transformer architecture in our experiments. In future work, it would be interesting to explore other transformer architectures to see if they can be used to approximate the matrix functions.

References

[1] Ben Adcock, Simone Brugiapaglia, and Clayton G Webster. *Sparse Polynomial Approximation of High-Dimensional Functions*, volume 25. SIAM, 2022.

[2] Ben Adcock, Simone Brugiapaglia, Nick Dexter, and Sebastian Moraga. Near-Optimal Learning of Banach-Valued, High-Dimensional Functions via Deep Neural Networks. *Neural Networks*, 181:106761, 2025.

[3] Mistral AI. Mixtral: A Mixture of Experts Model, 2023. URL `https://mistral.ai`. Large language model with mixture of experts.

[4] Dosovitskiy Alexey. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *arXiv preprint arXiv: 2010.11929*, 2020.

[5] Alberto Alfarano, François Charton, Amaury Hayat, and CERMICS-Ecole des Ponts Paristech. Discovering Lyapunov Functions with Transformers. In *The 3rd Workshop on Mathematical Reasoning and AI at NeurIPS*, volume 23, 2023.

[6] Alberto Alfarano, François Charton, and Amaury Hayat. Global Lyapunov Functions: A Long-Standing Open Problem in Mathematics, with Symbolic Transformers. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[7] Pedro Alonso, Javier Ibáñez, Jorge Sastre, Jesús Peinado, and Emilio Defez. Efficient and Accurate Algorithms for Computing Matrix Trigonometric Functions. *Journal of Computational and Applied Mathematics*, 309:325–332, 2017.

[8] Anthropic. Claude. `https://www.anthropic.com/claude`, 2024.

[9] Jimmy Lei Ba. Layer Normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[10] Alexei Baevski, Yuhao Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations. *Advances in Neural Information Processing Systems*, 33:12449–12460, 2020.

[11] GA Baker and P Graves-Morris. Padé Approximants 2nd edn. *Encyclopedia of Mathematics and its Applications, Scrics*, (59), 1996.

[12] Andrew R Barron. Universal Approximation Bounds for Superpositions of a Sigmoidal Function. *IEEE Transactions on Information theory*, 39(3):930–945, 1993.

[13] Richard Bellman. Dynamic Programming. *Science*, 153(3731):34–37, 1966.

[14] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The Long-Document Transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[15] Pierfrancesco Beneventano, Patrick Cheridito, Robin Graeber, Arnulf Jentzen, and Benno Kuckuck. Deep Neural Network Approximation Theory for High-Dimensional Functions. *arXiv preprint arXiv:2112.14523*, 2021.

[16] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning Long-Term Dependencies with Gradient Descent is Difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[17] Peter Benner, Enrique S Quintana-Ortí, and Gregorio Quintana-Ortí. Balanced Truncation Model Reduction of Large-Scale Dense Systems on Parallel Computers. *Mathematical and Computer Modelling of Dynamical Systems*, 6(4):383–405, 2000.

[18] Feliks Aleksandrovich Berezin and Mikhail Shubin. *The Schrödinger Equation*, volume 66. Springer Science & Business Media, 2012.

[19] Mogens Bladt and Bo Friis Nielsen. *Matrix-Exponential Distributions in Applied Probability*, volume 81. Springer, 2017.

[20] John S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. *Neurocomputing*, 68: 227–236, 1990.

[21] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[22] Shuhao Cao. Choose a Transformer: Fourier or Galerkin. *Advances in Neural Information Processing Systems*, 34:24924–24940, 2021.

[23] François Charton. What is my Math Transformer doing? – Three Results on Interpretability and Generalization. *arXiv preprint arXiv:2211.00170*, 2022.

[24] Francois Charton. Learning the Greatest Common Divisor: Explaining Transformer Predictions. In *The Twelfth International Conference on Learning Representations*, 2024.

[25] François Charton. Linear Algebra with Transformers. *arXiv preprint arXiv:2112.01898*, 12 2021. URL `http://arxiv.org/abs/2112.01898`.

[26] Alice Cortinovis, Daniel Kressner, and Stefano Massei. Divide-and-Conquer Methods for Functions of Matrices with Banded or Hierarchical Low-Rank Structure. *SIAM Journal on Matrix Analysis and Applications*, 43(1):151–177, 2022.

[27] George Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[28] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li,

Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2024. URL `https://arxiv.org/abs/2412.19437`.

[29] Li Deng, Geoffrey Hinton, and Brian Kingsbury. New Types of Deep Neural Network Learning for Speech Recognition and Related Applications: An Overview. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8599–8603. IEEE, 2013.

[30] Jacob Devlin. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[31] Fang Dong and Yinmei Lv. Matrix Operation and Its Application in Computer Engineering. In *2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS)*, pages 604–607. IEEE, 2018.

[32] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783*, 2024.

[33] Paul S Dwyer. Some Applications of Matrix Derivatives in Multivariate Analysis. *Journal of the American Statistical Association*, 62(318):607–625, 1967.

[34] Dennis Elbrächter, Dmytro Perekrestenko, Philipp Grohs, and Helmut Bölcskei. Deep Neural Network Approximation Theory. *IEEE Transactions on Information Theory*, 67(5):2581–2623, 2021.

[35] Jeffrey L Elman. Finding Structure in Time. *Cognitive Science*, 14(2):179–211, 1990.

[36] Luigi Fantappie. Le Calcul des Matrices. *CR Acad. Sci. Paris*, 186:619–621, 1928.

[37] Simon Foucart and Holger Rauhut. *A Mathematical Introduction to Compressive Sensing*. Birkhäuser Basel, 2013. ISBN 0817649476.

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

[39] Alex Graves and Alex Graves. Long Short-Term Memory. *Supervised Sequence Labelling with Recurrent Neural Networks*, pages 37–45, 2012.

[40] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, et al. Conformer: Convolution-Augmented Transformer for Speech Recognition. *arXiv preprint arXiv:2005.08100*, 2020.

[41] Jutho Haegeman, Christian Lubich, Ivan Oseledets, Bart Vandereycken, and Frank Verstraete. Unifying Time Evolution and Optimization with Matrix Product States. *Physical Review B*, 94(16):165116, 2016.

[42] PC Hammer. Adaptive Control Processes: A Guided Tour (R. Bellman), 1962.

[43] Joao P Hespanha. *Linear Systems Theory*. Princeton University Press, 2018.

[44] Nicholas J Higham. *Functions of Matrices: Theory and Computation*. SIAM, 2008.

[45] Nicholas J Higham. Siam's gene golub summer school. 2013. lecture 1 of: Functions of matrices., 2013. Accessed: September 02, 2024.

[46] Nicholas J Higham and Peter Kandolf. Computing the Action of Trigonometric and Hyperbolic Matrix Functions. *SIAM Journal on Scientific Computing*, 39(2):A613–A627, 2017.

[47] John J Hopfield. Neural Networks and Physical Systems with Emergent Collective Computational Abilities. *Proceedings of the National Academy of Sciences*, 79(8): 2554–2558, 1982.

[48] Roger A Horn and Charles R Johnson. *Matrix Analysis*. Cambridge university press, 2012.

[49] Kurt Hornik. Approximation Capabilities of Multilayer Feedforward Networks. *Neural networks*, 4(2):251–257, 1991.

[50] Steven Huss-Lederman, Elaine M Jacobson, Jeremy R Johnson, Anna Tsao, and Thomas Turnbull. Strassen's Algorithm for Matrix Multiplication: Modeling, Analysis, and Implementation. In *Proceedings of Supercomputing*, volume 96, pages 9–6. Citeseer, 1996.

[51] Sergey Ioffe. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv preprint arXiv:1502.03167*, 2015.

[52] Uday Kamath, John Liu, and James Whitaker. *Deep Learning for NLP and Speech Recognition*, volume 84. Springer, 2019.

[53] Charles S Kenney and Alan J Laub. The Matrix Sign Function. *IEEE Transactions on Automatic Control*, 40(8):1330–1348, 1995.

[54] ED Khoroshikh and VG Kurbatov. An Approximation of Matrix Exponential by a Truncated Laguerre Series. *arXiv preprint arXiv:2312.07291*, 2023.

[55] Diederik P Kingma. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[56] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The Efficient Transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[57] Andrei Nikolaevich Kolmogorov. On the Representation of Continuous Functions of Many Variables by Superposition of Continuous Functions of One Variable and Addition. In *Doklady Akademii Nauk*, volume 114, pages 953–956. Russian Academy of Sciences, 1957.

[58] Vaclav Kosar. Cross-attention in transformer architecture, 2024. URL `https://vaclavkosar.com/ml/cross-attention-in-transformer-architecture`. Accessed: September 02, 2024.

[59] Nesin Matematik Köyü. The Matrix Exponential. `https://nesinkoyleri.org/wp-content/uploads/2021/07/Exponential.pdf`, 2021. Lecture notes.

[60] Guillaume Lample and François Charton. Deep Learning for Symbolic Mathematics. *arXiv preprint arXiv:1912.01412*, 2019.

[61] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[62] Yann LeCun, Fu Jie Huang, and Leon Bottou. Learning Methods for Generic Object Recognition with Invariance to Pose and Lighting. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, volume 2, pages II–104. IEEE, 2004.

[63] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. Convolutional Networks and Applications in Vision. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 253–256. IEEE, 2010.

[64] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 521 (7553):436–444, 2015.

[65] Nayoung Lee, Kartik Sreenivasan, Jason D Lee, Kangwook Lee, and Dimitris Papailiopoulos. Teaching Arithmetic to Small Transformers. *arXiv preprint arXiv:2307.03381*, 2023.

[66] Zijie Li, Dule Shu, and Amir Barati Farimani. Scalable Transformer for PDE Surrogate Modeling. *Advances in Neural Information Processing Systems*, 36, 2024.

[67] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A Robustly Optimized BERT Pretraining Approach. *arXiv preprint arXiv:1907.11692*, 2019.

[68] Warren S McCulloch and Walter Pitts. A Logical Calculus of the Ideas Immanent in Nervous Activity. *The Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[69] Joseph McDonald, Baolin Li, Nathan Frey, Devesh Tiwari, Vijay Gadepally, and Siddharth Samsi. Great Power, Great Responsibility: Recommendations for Reducing Energy for Training Language Models. *arXiv preprint arXiv:2205.09646*, 2022.

[70] Hrushikesh N Mhaskar and Tomaso Poggio. Deep vs. Shallow Networks: An Approximation Theory Perspective. *Analysis and Applications*, 14(06):829–848, 2016.

[71] Anastasiia Minenkova, Evelyn Nitch-Griffin, and Vadim Olshevsky. Backward Stability of the Schur Decomposition under Small Perturbations. *Linear Algebra and its Applications*, 2024.

[72] Cleve Moler and Charles Van Loan. Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later. *SIAM review*, 20(4):801–836, 1978.

[73] Cleve Moler and Charles Van Loan. Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later. *SIAM Review*, 45(1):3–49, 2003.

[74] Vinod Nair and Geoffrey E Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[75] Kuniaki Noda, Yuki Yamaguchi, Kazuhiro Nakadai, Hiroshi G Okuno, and Tetsuya Ogata. Audio-Visual Speech Recognition Using Deep Learning. *Applied Intelligence*, 42:722–737, 2015.

[76] Rodrigo Nogueira, Zhiying Jiang, and Jimmy Lin. Investigating the Limitations of Transformers with Simple Arithmetic Tasks. *arXiv preprint arXiv:2102.13019*, 2021.

[77] OpenAI. ChatGPT (v. 4), 2023. URL `https://openai.com/chatgpt`. Large language model.

[78] Joost AA Opschoor, Ch Schwab, and Jakob Zech. Exponential ReLU DNN Expression of Holomorphic Maps in High Dimension. *Constructive Approximation*, 55(1):537–582, 2022.

[79] Daniel W Otter, Julian R Medina, and Jugal K Kalita. A Survey of the Usages of Deep Learning for Natural Language Processing. *IEEE Transactions on Neural Networks and Learning Systems*, 32(2):604–624, 2020.

[80] R Pascanu. On the Difficulty of Training Recurrent Neural Networks. *arXiv preprint arXiv:1211.5063*, 2013.

[81] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems*, 32, 2019.

[82] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon Emissions and Large Neural Network Training. *arXiv preprint arXiv:2104.10350*, 2021.

[83] Philipp Petersen and Felix Voigtlaender. Optimal Approximation of Piecewise Smooth Functions Using Deep ReLU Neural Networks. *Neural Networks*, 108:296–330, 2018.

[84] Stanislas Polu and Ilya Sutskever. Generative Language Modeling for Automated Theorem Proving. *arXiv preprint arXiv:2009.03393*, 2020.

[85] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal Mathematics Statement Curriculum Learning. *arXiv preprint arXiv:2202.01344*, 2022.

[86] Huizeng Qin and Youmin Lu. An Efficient Algorithm for Basic Elementary Matrix Functions with Specified Accuracy and Application. *Applied Mathematics*, 4(2):690–708, 2024.

[87] Jack Rae and et al. Gopher: A Scalable and Efficient Transformer for Text Generation. *arXiv preprint arXiv:2112.11446*, 2021. URL `https://arxiv.org/abs/2112.11446`.

[88] John Douglas Roberts. Linear Model Reduction and Solution of the Algebraic Riccati Equation by Use of the Sign Function. *International Journal of Control*, 32(4):677–687, 1980.

[89] Sheldon M Ross. *Introduction to Probability Models*. Academic Press, 2014.

[90] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation, parallel distributed processing, explorations in the microstructure of cognition, ed. de rumelhart and j. mcclelland. vol. 1. 1986. *Biometrika*, 71(599-607):6, 1986.

[91] Michael E Sander, Pierre Ablin, Mathieu Blondel, and Gabriel Peyré. Sinkformers: Transformers with Doubly Stochastic Attention. In *International Conference on Artificial Intelligence and Statistics*, pages 3515–3530. PMLR, 2022.

[92] Clayton Sanford, Daniel J Hsu, and Matus Telgarsky. Representational Strengths and Limitations of Transformers. *Advances in Neural Information Processing Systems*, 36, 2024.

[93] LS Shieh, YT Tsay, and CT Wang. Matrix Sector Functions and Their Applications to Systems Theory. In *IEE Proceedings D (Control Theory and Applications)*, volume 131, pages 171–181. IET, 1984.

[94] Ralph C Smith. *Uncertainty Quantification: Theory, Implementation, and Applications*. SIAM, 2024.

[95] Richard Socher, Yoshua Bengio, and Christopher D Manning. Deep Learning for NLP (Without Magic). In *Tutorial Abstracts of ACL 2012*, pages 5–5. 2012.

[96] Danny C Sorensen and Yunkai Zhou. Direct Methods for Matrix Sylvester and Lyapunov Equations. *Journal of Applied Mathematics*, 2003(6):277–303, 2003.

[97] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[98] Ilya Sutskever, Oriol Vinyals, and Quoc Le. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*, 2014.

[99] Matthew Tancik, Pratul Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan Barron, and Ren Ng. Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains. *Advances in Neural Information Processing Systems*, 33:7537–7547, 2020.

[100] Tina Torabi, Timon S Gutleb, and Christoph Ortner. Fast Automatically Differentiable Matrix Functions and Applications in Molecular Simulations. *arXiv preprint arXiv:2412.12598*, 2024.

[101] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models, 2023. URL `https://arxiv.org/abs/2302.13971`.

[102] Alan M Turing. *Computing Machinery and Intelligence*. Springer, 2009.

[103] Ashish Vaswani. Attention is All You Need. *arXiv preprint arXiv:1706.03762*, 2017.

[104] Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-Attention with Linear Complexity. *arXiv preprint arXiv:2006.04768*, 2020.

[105] Yu Zhang, Peter Tiňo, Aleš Leonardis, and Ke Tang. A Survey on Neural Network Interpretability. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 5(5):726–742, 2021.

[106] Zixing Zhang, Jürgen Geiger, Jouni Pohjalainen, Amr El-Desoky Mousa, Wenyu Jin, and Björn Schuller. Deep Learning for Environmentally Robust Speech Recognition: An Overview of Recent Developments. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 9(5):1–28, 2018.

Appendix A

Appendix For Chapter 2

We describe some preliminaries and definitions used in Chapter 2.

Definition A.0.1 (Matrix Similarity Transformation [48])**.** *Two square matrices A and B of size $n \times n$ are said to be* ***similar*** *if there exists an invertible matrix P such that:*

$$B = P^{-1}AP.$$

In this case, we write:

$$A \sim B.$$

The following function application to similar matrices is fundamental in matrix function computations.

Definition A.0.2 (Function Application to Similar Matrices)**.** *If f is a matrix function (e.g., exponentials, polynomials, or trigonometric functions), then applying f to a similarity transformation follows:*

$$f(B) = f(P^{-1}AP).$$

For many analytic functions defined via power series, this simplifies to:

$$f(B) = P^{-1}f(A)P.$$

This property is fundamental in matrix function computations, as it ensures that similar matrices maintain the same spectral characteristics under function application.

We denote the notation of $\lesssim$ in the following definition.

Definition A.0.3 (Notation $\lesssim$ [44])**.** *Let a, b be real numbers or functions, or let A, B be matrices with a chosen norm $\|\cdot\|$. The notation*

$$a \lesssim b,$$

or

$$A \lesssim B,$$

means that there exists a constant $C > 0$ such that

$$a \leq Cb, \quad \text{or} \quad \|A\| \leq C\|B\|.$$

The constant C is typically independent of key parameters in the problem but may depend on structural properties of the space or the specific norm used.

Appendix B

Appendix For Chapter 3

This chapter introduces certain concepts key to understanding the mathematics of neural networks. Non linearities that help the neural network learn complex patterns are introduced in the following definitions.

Definition B.0.1 (Softmax Function)**.** *The softmax function [20] converts a vector of real numbers into a probability distribution:*

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}, \quad \text{for } i = 1, 2, \ldots, n.$$

It ensures that each output is in the range $(0, 1)$ *and that the sum of all outputs equals 1.*

The ReLU [74] function is a popular non-linear activation function in neural networks.

Definition B.0.2 (ReLU Function)**.** *The Rectified Linear Unit (ReLU) activation function [74] is defined as:*

$$\text{ReLU}(x) = \max(0, x).$$

For positive inputs, it outputs x*, while for non-positive inputs, it outputs 0.*

The loss function is a measure that quantifies the difference between the predicted output and the true target. We introduce the following definition.

Definition B.0.3 (Loss Function)**.** *A loss function measures the difference between the true target* y *and the predicted output* $\hat{y}$*. It is denoted as:*

$$\mathcal{L}(y, \hat{y}) = \mathcal{L}(y, f(x; \theta))$$

where $f(x; \theta)$ *represents the model's prediction.*

B.1 Layer Normalization

Layer normalization is a technique used to normalize the activations within each individual data sample instead of across a batch. We define the following.

Definition B.1.1 (Layer Normalization (LN))**.** *Layer Normalization (LN) [9] normalizes the activations within each individual data sample instead of across a batch. Given an input* $x \in \mathbb{R}^d$*:*

1. *Compute the mean:*

$$\mu = \frac{1}{d} \sum_{i=1}^{d} x_i$$

2. *Compute the variance:*

$$\sigma^2 = \frac{1}{d} \sum_{i=1}^{d} (x_i - \mu)^2$$

3. *Normalize:*

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

4. *Apply learnable scale and shift parameters:*

$$y_i = \gamma \hat{x}_i + \beta$$

where γ and β are learnable parameters.

B.2 Feedforward Neural Network

A **Feedforward Neural Network (FNN)** is a function $f : \mathbb{R}^{d_0} \to \mathbb{R}^{d_L}$ that maps an input $x \in \mathbb{R}^{d_0}$ to an output using a sequence of affine transformations followed by nonlinear activation functions. [38] The network consists of L layers: an input layer, hidden layers, and an output layer. An more comprehensive definition can be found in [38, 49].

Mathematical Representation:

- **Input Layer:**

$$h_0 = x \in \mathbb{R}^{d_0}$$

- **Hidden Layers** (for $\ell = 1, \ldots, L-1$):

$$h_\ell = \sigma(W_\ell h_{\ell-1} + b_\ell), \quad h_\ell \in \mathbb{R}^{d_\ell}$$

  where:

  - $W_\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ is the weight matrix,
  - $b_\ell \in \mathbb{R}^{d_\ell}$ is the bias vector,
  - $\sigma : \mathbb{R} \to \mathbb{R}$ is an element-wise nonlinear activation function (e.g., ReLU, Sigmoid).

- **Output Layer:**

$$f(x) = W_L h_{L-1} + b_L.$$

  If a final activation function ϕ is used (e.g., Softmax in classification problems), then:

$$f(x) = \phi(W_L h_{L-1} + b_L).$$

B.3 Recurrent Neural Network (RNN)

Recurrent neural networks (RNNs) are a type of neural network that processes sequential data by maintaining a hidden state that captures information from previous time steps. We define the following.

Definition B.3.1 (Recurrent Neural Network (RNN))**.** *A recurrent neural network (RNN) [90] by Rumelhart et al. (1986) is a type of neural network that processes sequential data by maintaining a hidden state that captures information from previous time steps.*

Mathematical Formulation:

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1} + b_{hh}) \; y_t \qquad = W_{ho}h_t + b_{ho}$$

where tanh is the hyperbolic tangent function, W_{ih} and W_{hh} are weight matrices, b_{ih} and b_{hh} are bias vectors, and W_{ho} and b_{ho} are output weight and bias vectors. Recurrent networks typically produce an output at each time step and have recurrent connections between hidden units [38].

B.4 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) that can capture long-term dependencies in sequential data. We define the following.

Definition B.4.1 (Long Short-Term Memory (LSTM))**.** *LSTMs [39] are a type of recurrent neural network (RNN) that can capture long-term dependencies in sequential data. Each LSTM cell consists of a memory cell C_t and three gates: forget, input, and output.*

Mathematical Formulation:

$$\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \quad \textit{(Forget Gate)} \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \quad \textit{(Input Gate)} \\
\tilde{C}_t &= \tanh(W_C x_t + U_C h_{t-1} + b_C) \quad \textit{(Candidate Cell State)} \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \quad \textit{(Cell State Update)} \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \quad \textit{(Output Gate)} \\
h_t &= o_t \odot \tanh(C_t) \quad \textit{(Hidden State)}
\end{aligned}$$

where σ is the sigmoid function and $\odot$ represents element-wise multiplication.

Appendix C

Appendix For Chapter 4

This chapter introduces the theorems and propositions used in Chapter 4.

C.1 Theorem: DNN Architecture for Matrix Exponential

The following theorem is taken from Higham (2008) (see [44] for more details).

Theorem C.1.1 (Convergence of Matrix Taylor Series)**.** *Suppose f has a Taylor series expansion*

$$f(z) = \sum_{k=0}^{\infty} a_k (z-\alpha)^k \qquad \left(a_k = \frac{f^{(k)}(\alpha)}{k!} \right),$$

with radius of convergence r. If $A \in \mathbb{C}^{n\times n}$ then $f(A)$ is defined and is given by

$$f(A) = \sum_{k=0}^{\infty} a_k (A-\alpha I)^k,$$

if and only if each of the distinct eigenvalues $\lambda_1, \ldots, \lambda_s$ of A satisfies one of the conditions:

1. *$|\lambda_i - \alpha| < r$,*
2. *$|\lambda_i - \alpha| = r$ and the series for $f^{(n_i-1)}(\lambda)$ (where n_i is the index of λ_i) is convergent at the point $\lambda = \lambda_i$, $i = 1:s$.*

The following theorem provides an estimate of the truncation error which we used in Theorem 4.3.1 when approximating matrix functions using Taylor series.

Theorem C.1.2 (Taylor Series Truncation Error Bound)**.** *For a matrix function f with Taylor series expansion (4.6) with radius of convergence r, if $A \in \mathbb{C}^{n\times n}$ with $\rho(A-\alpha I) < r$ then for any matrix norm*

$$\left\| f(A) - \sum_{k=0}^{s} \frac{f^{(k)}(\alpha)}{k!} (A-\alpha I)^k \right\| \le \frac{1}{s!} \max_{0\le t\le 1} \|(A-\alpha I)^{s+1} f^{(s+1)}(\alpha I + t(A-\alpha I))\|.$$

This bound from Higham (2008) (see [44]) provides an estimate of the truncation error when approximating matrix functions using Taylor series.

The next two definitions are taken from Petersen and Voigtlaender (2018) (see [83] for more details). These theorems outline the concatenation and parallelization of neural networks.

Theorem C.1.3 (Concatenation of Neural Networks)**.** *Let $L_1, L_2 \in \mathbb{N}$ and let*

$$\Phi^1 = \Big((A_1^1, b_1^1), \dots, (A_{L_1}^1, b_{L_1}^1)\Big)$$

and

$$\Phi^2 = \Big((A_1^2, b_1^2), \dots, (A_{L_2}^2, b_{L_2}^2)\Big)$$

be two neural networks such that the input layer of Φ^1 has the same dimension as the output layer of Φ^2. Then, $\Phi^1 \bullet \Phi^2$ denotes the following $L_1 + L_2 - 1$ layer network:

$$\Phi^1 \bullet \Phi^2 := \Big((A_1^2, b_1^2), \dots, (A_{L_2-1}^2, b_{L_2-1}^2), (A_1^1 A_{L_2}^2, A_1^1 b_{L_2}^2 + b_1^1),$$
$$(A_2^1, b_2^1), \dots, (A_{L_1}^1, b_{L_1}^1)\Big).$$

We call $\Phi^1 \bullet \Phi^2$ the concatenation of Φ^1 and Φ^2.

One directly verifies that

$$R_\varrho(\Phi^1 \bullet \Phi^2) = R_\varrho(\Phi^1) \circ R_\varrho(\Phi^2),$$

which shows that the definition of concatenation is reasonable.

If the activation function $\varrho : \mathbb{R} \to \mathbb{R}$ is the ReLU – that is,

$$\varrho(x) = \max\{0, x\}$$

– then, based on the identity $x = \varrho(x) - \varrho(-x)$ for $x \in \mathbb{R}$, one can construct a simple two-layer network whose realization is the identity $\mathrm{Id}_{\mathbb{R}^d}$ on $\mathbb{R}^d$.

Theorem C.1.4 (Parallelization of Neural Networks)**.** *Let $L \in \mathbb{N}$ and let $\Phi^1 = ((A_1^1, b_1^1), \dots, (A_L^1, b_L^1))$ and $\Phi^2 = ((A_1^2, b_1^2), \dots, (A_L^2, b_L^2))$ be two neural networks with L layers and with d-dimensional input. We define*

$$P(\Phi^1, \Phi^2) := ((\tilde{A}_1, \tilde{b}_1), \dots, (\tilde{A}_L, \tilde{b}_L)),$$

where

$$\tilde{A}_1 := \begin{pmatrix} A_1^1 \\ A_1^2 \end{pmatrix}, \quad \tilde{b}_1 := \begin{pmatrix} b_1^1 \\ b_1^2 \end{pmatrix} \quad \text{and}$$

$$\tilde{A}_\ell := \begin{pmatrix} A_\ell^1 & 0 \\ 0 & A_\ell^2 \end{pmatrix}, \quad \tilde{b}_\ell := \begin{pmatrix} b_\ell^1 \\ b_\ell^2 \end{pmatrix} \quad \text{for } 1 < \ell \leq L.$$

Then, $P(\Phi^1, \Phi^2)$ is a neural network with d-dimensional input and L layers, called the parallelization of Φ^1 and Φ^2.

One readily verifies that $M(P(\Phi^1, \Phi^2)) = M(\Phi^1) + M(\Phi^2)$, and

$$R_\varrho(P(\Phi^1, \Phi^2))(x) = (R_\varrho(\Phi^1)(x), R_\varrho(\Phi^2)(x))$$

for all $x \in \mathbb{R}^d$.

The next proposition is taken from Opschoor et al. (2022) (see [78] for more details). This proposition outlines the multiplication of two neural networks.

Proposition C.1.5 ([78, Proposition 3.1]). *For any $\delta \in (0,1)$ and $M \geq 1$ there exists a σ_1-NN $\tilde{\times}_{\delta,M} : [-M,M]^2 \to \mathbb{R}$ such that*

$$\sup_{|a|,|b|\leq M} |ab - \tilde{\times}_{\delta,M}(a,b)| \leq \delta,$$

$$\text{ess sup}_{|a|,|b|\leq M} \max\left\{\left|b - \frac{\partial}{\partial a}\tilde{\times}_{\delta,M}(a,b)\right|, \left|a - \frac{\partial}{\partial b}\tilde{\times}_{\delta,M}(a,b)\right|\right\} \leq \delta,$$

where $\frac{\partial}{\partial a}\tilde{\times}_{\delta,M}(a,b)$ and $\frac{\partial}{\partial b}\tilde{\times}_{\delta,M}(a,b)$ denote weak derivatives. There exists a constant $C > 0$ independent of $\delta \in (0,1)$ and $M \geq 1$ such that $\text{size}_{in}(\tilde{\times}_{\delta,M}) \leq C$, $\text{size}_{out}(\tilde{\times}_{\delta,M}) \leq C$,

$$\text{depth}(\tilde{\times}_{\delta,M}) \leq C(1 + \log_2(M/\delta)), \quad \text{size}(\tilde{\times}_{\delta,M}) \leq C(1 + \log_2(M/\delta)).$$

Moreover, for every $a \in [-M,M]$, there exists a finite set $\mathcal{N}_a \subseteq [-M,M]$ such that $b \mapsto \tilde{\times}_{\delta,M}(a,b)$ is strongly differentiable at all $b \in (-M,M) \setminus \mathcal{N}_a$.

Appendix D

Appendix For Chapter 5

This chapters introduces the shaded plots used in Chapter 5 in Section D.1. It is for the reader to have a better understanding of the plots in Chapter 5. We also introduce the L1 norm (Manhattan distance) in Section D.2.

D.1 Shaded Plots

This section is taken from Adcock et al. (2022) (see [1, Appendix A.1.3] for more details).

For experiments that involve statistical simulations, we perform multiple random trials (i.e., random draws of the sample points) and display certain statistics of the sampled variable of interest. Throughout, we set the number of trials to be

$$N_{\text{trials}} = 50$$

Suppose that the hyperparameter values considered in a certain simulation are

$$x_i, \quad i = 1, \dots, n$$

We denote the simulated samples of the variable of interest as

$$y_i^{(k)}, \quad i = 1, \dots, n, k = 1, \dots, N_{\text{trial}}$$

In the settings of our experiments, we often observe large variations of the variable of interest as a function of the hyperparameter; typically, by orders of magnitude. Hence, the variable of interest is always measured using a base-10 logarithmic scale in the y axis. For this reason, it is natural to visualize statistics of the variable of interest after a logarithmic transformation. For every value of $i = 1, \dots, n$, we compute the sample mean and the (corrected) sample standard deviation of the transformed sample

$$\left\{\log_{10}\left(y_i^{(k)}\right)\right\}_{k=1}^{N_{\text{trial}}}$$

Namely, for every $i = 1, \dots, n$, we compute

$$\mu_i = \frac{1}{N_{\text{trial}}} \sum_{k=1}^{N_{\text{trial}}} \log_{10}\left(y_i^{(k)}\right) \quad \text{and} \quad \sigma_i = \sqrt{\frac{1}{N_{\text{trial}} - 1} \sum_{k=1}^{N_{\text{trial}}} \left(\log_{10}\left(y_i^{(k)}\right) - \mu_i\right)^2}$$

These statistics of the variable of interest are visualized using shaded plots. They are formed by two main components: 1. A main curve obtained by plotting the data points $\{(x_i, 10^{\mu_i})\}_{i=1}^{n}$, which represents the average behaviour (in logarithmic scale) of the variable of interest. 2. A shaded region bounded by a lower and an upper curve, obtained by plotting $\{(x_i, 10^{\mu_i - \sigma_i})\}_{i=1}^{n}$ and $\{(x_i, 10^{\mu_i + \sigma_i})\}_{i=1}^{n}$, respectively. The width of this region represents the amount of dispersion of the variable of interest.

Notice that 10^{μ_i} is precisely the geometric mean of the values $\left\{y_i^{(k)}\right\}_{k=1}^{N_{\text{trial}}}$ and 10^{σ_i} its (corrected) geometric standard deviation.

D.2 ℓ_1 Norm (Manhattan Distance)

This section introduces the ℓ_1 norm (Manhattan distance) which is used in Chapter 5.

Definition D.2.1 (ℓ_1 Norm (Manhattan Distance))**.** *The ℓ_1 norm (also known as Manhattan distance or taxicab norm) of a vector $x \in \mathbb{R}^n$ is defined as:*

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|$$

where $|x_i|$ denotes the absolute value of the i-th component. The name Manhattan distance comes from the distance a taxi would drive in a city laid out in a grid-like pattern (like Manhattan), where the taxi can only drive along the grid lines.