Arithmetic Circuits, Structured Matrices and (not so) Deep Learning

ATRI RUDRA

Department of Computer Science and Engineering
University at Buffalo
atri@buffalo.edu

Abstract

This survey presents a necessarily incomplete (and biased) overview of results at the intersection of arithmetic circuit complexity, structured matrices and deep learning. Recently there has been some research activity in replacing unstructured weight matrices in neural networks by structured ones (with the aim of reducing the size of the corresponding deep learning models). Most of this work has been experimental and in this survey, we formalize the research question and show how a recent work that combines arithmetic circuit complexity, structured matrices and deep learning essentially answers this question.

This survey is targeted at complexity theorists who might enjoy reading about how tools developed in arithmetic circuit complexity helped design (to the best of our knowledge) a new family of structured matrices, which in turn seem well-suited for applications in deep learning. However, we hope that folks primarily interested in deep learning would also appreciate the connections to complexity theory.

IN MEMORY OF ALAN SELMAN

Alan Selman was my colleague at University at Buffalo (UB) from 2007 (when I joined UB) until 2014 (when Alan retired from UB). I still remember being taken directly from the airport to Alan's favorite restaurant, Trattoria Aroma, during my interview at Buffalo. I was a bit intimidated by Alan during the dinner but we bonded over the fact that we were both married to epidemiologists. After I joined Buffalo, Alan's sage advice helped me throughout my tenure process. Alan was a giant in the department and having him in my corner did not hurt.

More germane to this survey, Alan always turned up for UB theory meetings and I greatly enjoyed presenting stuff I was working on to Alan during some of these meetings. After Alan retired in 2014, I started working on some problems at the intersection of arithmetic circuit complexity, structured matrices and deep learning that I think Alan would be enjoyed hearing about. Since Alan passed in early 2021, this survey is my way of presenting the material to Alan in his memory. – ATRI RUDRA

1 Introduction

This survey shows how concepts in arithmetic circuit complexity and structured matrices can be used to solve a (theoretical) problem motivated by practical applications in machine learning (especially deep learning). Since each of the areas of arithmetic (circuit) complexity, structured matrices and deep learning have been explored in great depth and this survey clearly cannot do any justice to all the great work in each of the these areas, we will spend most of the introduction clarifying what this survey is *not* about.

Algebraic circuit complexity or more generally algebraic complexity theory [11] studies the power of algebraic algorithms (as opposed to the Turing machine/RAM model). The arithmetic circuit model (or the straight-line programs) are one of the standard models of computation in algebraic complexity theory [11, Chapter 4]. In this survey we will ignore pretty much everything in this literature except for results on the arithmetic circuit complexity of the linear map i.e. functions of the form $\boldsymbol{x} \mapsto \mathbf{W}\boldsymbol{x}$ (where $\boldsymbol{x}$ is a vector over some field $\mathbb{F}$ and $\mathbf{W}$ is a matrix over the same field) [11, Chapter 13]. We would like to stress that this survey will only scratch the surface of the literature on the algebraic circuit complexity of the linear map. Just to give a sense of the breadth of this seemingly 'specialized' topic, we remark that the study of *matrix rigidity* [22], which has seen a lot of recent research activity [1, 2, 3, 19, 10], is a part of this topic. We note that originally, the topic of matrix rigidity was proposed by Valiant [41] as a way to prove super-linear lower bounds, by constructing matrices that are *rigid*. However, our goal in this survey is to prove upper bounds– i.e. we are interested in matrices for which the arithmetic circuit complexity is small. We note that some of the recent work, including the work of Alman and Williams [1], is along similar lines of showing that explicit matrices are *not* rigid (which at a very hand-wavy level is showing that for certain explicit linear maps there indeed exist 'small' arithmetic of a *restricted kind* to compute the linear map[1] – see Section 4.3 for more details).

Structured matrices are (family) of matrices $\mathbf{W}$ for which one can have a much smaller representation than the generic $n \times n$ (assuming $\mathbf{W}$ is a square) matrix representation. Typically, these structured representations also imply that one can compute $\mathbf{W}\boldsymbol{x}$ for any vector $\boldsymbol{x}$ in $o(n^2)$ time (recall that matrix-vector

[1] The notion of small here is to show circuits of size $o(n^2)$ but the bounds are still $\Omega\left(n^{2-\varepsilon}\right)$ for any fixed $\varepsilon > 0$, while in our case we are more interested in linear maps that have a near-linear sized *general* arithmetic circuits.

multiplication in the worst-case takes $O(n^2)$ time), and in many celebrated examples (e.g. FFT for the Discrete Fourier matrix [14]), it takes near-linear time/operations over the underlying field. At the risk of over-simplifying things, structured matrices crop up in applications in two flavors. In the first flavor, the application essentially determines the (family) of structured matrices. In other words, we do not have any say in the choice of the structured matrix and the goal is design efficient matrix vector multiplication algorithm (or algorithm for some other problems) involving the matrix. We mention two examples. The first example is the family of *orthogonal polynomial transforms* (which include among others the Discrete Cosine Transform) [37] that appear in many signal processing applications as well as many basic mathematical studies including approximation theory. The second example is the family of *low displacement rank* matrices [25], which have applications in signal processing and numerical linear algebra [26]. We will not cover this flavor of structured matrices in the survey though low displacement rank matrices will make an appearance in Section 4.5.

The second flavor of structured matrices (which we will focus on in this survey), is where there is some matrix $\mathbf{M}$ in the 'wild' and we want to approximate $\mathbf{M}$ by a more structured matrix $\mathbf{W}$ to e.g. save on storing the matrix (and/or have more efficient operations on the matrix: e.g. matrix vector multiplication). Perhaps *the* example of this is the ubiquitous low rank approximation. Udell and Townsend give a theoretical justification for why low rank approximation is so ubiquitous in machine learning applications [40]. Now we present a (very incomplete) sampler of other applications of structured matrices in machine learning– convolutions for image, language, and speech modeling [23], and low-rank and sparse matrices for efficient storage and inference on edge devices [42]. Forms of structure such as sparsity have been at the forefront of recent advances in machine learning [21], and are critical for on-device and energy-efficient models, two application areas of tremendous recent interest [39, 35].

At a very high level, the main question we consider in this survey is if there is a similar family of structured matrices that has all the nice properties of low rank approximation but are more expressive than low rank matrices (e.g. many of the transforms including the Fourier transform are full rank).

Deep learning is ubiquitous in our daily lives [28] with far reaching consequences– both good and bad[2]. For this survey, we will focus on the mathematical aspects of deep learning since a treatment of the societal implications of deep learning is out of the scope of this survey. Even a broad theoretical study of neural networks (which form the basis of deep learning) is beyond the scope of this survey and there is a lot of excellent literature on this topic [4] that we will side-step.

Instead, we will focus on the issue that deep learning models are getting to be too big (which in parallel raises[3] its own ethical issues [7]). This for example, can be an issue when trying to store these models (and run inference) on mobile platforms like smartphones. In addition, the state-of-the-art language models have so many parameters that creating such models is not possible outside of large technology companies. While there are many reasons for this, *one typical reason* is that these neural networks tend to learn *un*structured matrices as part of the neural network model (see Section 2.3 for why matrices make an appearance in neural network architectures). Apriori, the advantage of learning from the set of all possible matrices is that it gives the training algorithm the 'best' chance to learn the most expressive matrix. However, given that in many situations there is a budget on how many parameters we can use in representing the matrices, the high level question we consider in this survey is:

[2]This has led to deep intellectual research on societal implications of machine learning even in the theory community [5].

[3]OK, we could not resist. This though is the last mention of societal issues in the survey.

Question 1.1. *Given a budget on number of parameters that one can use to represent a matrix, what is the 'most expressive' family of matrices?*

We remark that a *lot* of recent innovations in deep learning have come from designing new architectures of neural networks, which needless to say, is out of scope for the survey (and the author!). In particular, in this survey we will consider a toy version of a *single layer* neural network, which by definition is *not so deep*.

Organization of the survey. We present some preliminaries and background before formalizing Question 1.1 in Section 2. We also formalize the problem of training a neural network (the Baur-Strassen theorem [6] plays a starring role) in Section 3. In Section 4, we analyze existing families of structured matrices and show how they all fall short in answering Question 1.1 (or more precisely its formal version Question 2.3). In Section 5, we survey results from Dao et al. [17] who present (to the best of our knowledge) a new family of structured matrices that indeed answers Question 2.3 in the affirmative. We conclude with a (biased) list of open questions in Section 6.

2 Preliminaries and Problem Definition

We begin by setting up notation in Section 2.1. We setup necessary background in Section 2.2 (matrix vector multiplication), Section 2.3 (neural networks), Section 2.4 (structured matrices) and Section 2.5 (arithmetic circuits). Finally, we formalize Question 1.1 in Section 2.6.

2.1 Notation

We use $\mathbb{F}$ to denote a field[4]. The set of all length n vectors and $m \times n$ matrices over $\mathbb{F}$ are denoted by $\mathbb{F}^n$ and $\mathbb{F}^{m\times n}$ respectively.

We will denote the entry in $\mathbf{W} \in \mathbb{F}^{m\times n}$ corresponding to the ith row and jth column as $\mathbf{W}[i,j]$. The ith row of $\mathbf{W}$ will be denoted by $\mathbf{W}[i,:]$. Similarly, the ith entry in the vector $\boldsymbol{x}$ will be denoted as $\boldsymbol{x}[i]$. We will follow the convention that the indices i and j start at 0. The inner product of vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ will be denoted by $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$. For any $\boldsymbol{x} \in \mathbb{F}^n$, we will use $\mathrm{diag}(\boldsymbol{x})$ to denote the diagonal matrix with $\boldsymbol{x}$ being its diagonal.

We will be using asymptotic notation and use $\widetilde{O}(\cdot)$ to hide poly-log factors in the Big-Oh notation.

2.2 Matrix Vector Multiplication

We now define the matrix-vector multiplication problem that will be central to the survey:

- **Input**: An $m \times n$ matrix $\mathbf{W} \in \mathbb{F}^{m\times n}$ and a vector $\boldsymbol{x} \in \mathbb{F}^n$ of length n
- **Output**: Their product, which is denoted by

$$\boldsymbol{y} = \mathbf{W} \cdot \boldsymbol{x},$$

[4] We will pretty much use $\mathbb{F} = \mathbb{R}$ (real number) or $\mathbb{F} = \mathbb{C}$ (complex numbers) in the survey. Even though most of the results in the survey can be made to work for finite fields, we will ignore this aspect of the results.

where $\boldsymbol{y} \in \mathbb{F}^m$ is a vector of length m and its ith entry for $0 \le i < m$ is defined as follows:

$$\boldsymbol{y}[i] = \sum_{j=0}^{n-1} \mathbf{W}[i,j] \cdot \boldsymbol{x}[j].$$

One can easily verify that the naive algorithm that basically operationalizes the above definition takes $O(mn)$ operations in the worst-case. Further, if the matrix $\mathbf{W}$ is arbitrary, one would need $\Omega(mn)$ time (this follows from a simple adversarial argument). Assuming that each operation for an field $\mathbb{F}$ can be done in $O(1)$ time, this implies that the worst-case complexity of matrix-vector multiplication is $\Theta(mn)$.

If we just cared about worst-case complexity, we would be done. However, since there is a fair bit of survey left after this spot it is safe to assume that this is not all we care about. It turns out that in a large number of practical applications, the matrix $\mathbf{W}$ is fixed (or more appropriately has some structure). Thus, when designing algorithms to compute $\mathbf{W} \cdot \boldsymbol{x}$ (for arbitrary $\boldsymbol{x}$), we can exploit the structure of $\mathbf{W}$ to obtain a complexity that is asymptotically better than $O(mn)$.

Next, we take a brief detour into deep learning and motivate why one would need structured matrices in that application.

2.3 Neural Networks and (not so) deep learning

WARNING: We do not claim to have any non-trivial knowledge (deep or otherwise) of deep learning. Thus, we will only consider a very simplified model of neural networks and our treatment of neural networks should in no way be interpreted as being representative of the current state of deep learning.

We consider a toy version of neural networks in use today: we will consider the so called *single layer* neural network:

Definition 2.1. *We define a* single layer neural network *with input $\boldsymbol{x} \in \mathbb{F}^n$ and output $\boldsymbol{y} \in \mathbb{F}^m$ where the output is related to input as follows:*

$$\boldsymbol{y} = g(\mathbf{W} \cdot \boldsymbol{x}),$$

where $\mathbf{W} \in \mathbb{F}^{m \times n}$ and $g : \mathbb{F}^m \to \mathbb{F}^m$ is a non-linear function.

Some remarks are in order: (1) In practice, neural networks are defined for $\mathbb{F} = \mathbb{R}$ or $\mathbb{F} = \mathbb{C}$; (2) One of the common examples of non-linear function $g : \mathbb{R}^m \to \mathbb{R}^m$ is applying to so called ReLu function to each entry.[5] (3) The entries in the matrix $\mathbf{W}$ are typically called the *weights* in the layer.

Neural networks have two tasks associated with it: the first is the task of learning the network. For the network in Definition 2.1, this implies learning the matrix $\mathbf{W}$ given a set of training data $(\boldsymbol{x}_0, \boldsymbol{y}_0), (\boldsymbol{x}_1, \boldsymbol{y}_1), \cdots$ where $\boldsymbol{y}_i$ is supposed to be a noisy version of $g(\mathbf{W}\boldsymbol{x})$– we will come back to this in Section 3.1.

The second task is that once we have learned $\mathbf{W}$, we use it to *classify* new data points $\boldsymbol{x}$ by computing $g(\mathbf{W}\boldsymbol{x})$. In practice, we would like the second step to be as efficient as possible.[6] Ideally we should be able to compute $g(\mathbf{W}\boldsymbol{x})$ with $O(m+n)$ operations. The computational bottleneck in computing $g(\mathbf{W}\boldsymbol{x})$ is computing $\mathbf{W} \cdot \boldsymbol{x}$. Further, it turns out (as well will see later in Section 3) that the complexity of the first step of learning the network is closely related to the complexity of the corresponding matrix-vector multiplication problem.

[5] More precisely, we have $\mathsf{ReLu}(x) = \max(0, x)$ for any $x \in \mathbb{R}$ and for any $\boldsymbol{z} \ni \mathbb{R}^m$, $g(\boldsymbol{z}) = (\mathsf{ReLu}(\boldsymbol{z}[0]), \cdots, \mathsf{ReLu}(\boldsymbol{z}[m-1]))$.

[6] Ideally, we would also like the first step to be efficient but typically the learning of the network can be done in an offline step so it can be (relatively) more inefficient.

2.4 Structured Matrices

As mentioned above, in the deep learning setup, we would like to have weight matrices $\mathbf{W}$ such that the matrix-vector multiplication $\mathbf{W}\boldsymbol{x}$ for an arbitrary $\boldsymbol{x} \in \mathbb{F}^n$ can be done in near-linear time. However, if the matrix $\mathbf{W}$ is represented in the usual $m \times n$ matrix format, then we end up with an $\Omega(mn)$ time just to read the entries of $\mathbf{W}$. Thus, to have any hope of near-linear matrix vector multiplication, we need to have a smarter representation of the structured matrices. We first recall two examples of structured matrices that have wide applicability in numerical linear algebra and machine learning.

We begin with the notion of low-rank matrices (which are ubiquitous in machine learning [40]):

Definition 2.2 (Low rank matrices)**.** *A matrix $\mathbf{W} \in \mathbb{F}^{m \times n}$ has rank r (for $0 \leq r \leq \min(m,n)$) if and only if there exists matrices $\mathbf{L} \in \mathbb{F}^{m \times r}$ and $\mathbf{R} \in \mathbb{F}^{r \times n}$ such that*

$$\mathbf{W} = \mathbf{L} \cdot \mathbf{R}.$$

It is easy to see that rank r matrices can be represented in $r(n+m)$ elements (by storing $\mathbf{L}$ and $\mathbf{R}$) and also has an $O(r(n+m))$-operations matrix vector multiplication by computing $\mathbf{W}\boldsymbol{x}$ as $(\mathbf{L} \cdot (\mathbf{R} \cdot \boldsymbol{x}))$. Thus, constant rank matrices indeed satisfy the linear-time matrix-vector multiplication desiderata.

Next, we consider sparse matrices:

Definition 2.3 (Sparse matrices)**.** *A matrix $\mathbf{W} \in \mathbb{F}^{m \times n}$ is s sparse (for $0 \leq s \leq mn$) if at most s entries in $\mathbf{W}$ are non-zero.*

The defacto representation of sparse matrices is the *listing representation*, where one keeps a list of the locations of the s non-zero values along with the actual non-zero value (every entry not in this list has a value of 0). Assuming the listing representation, the obvious modification to the naive matrix-vector multiplication (where we automatically 'skip' over entries (i,j) such that $\mathbf{W}[i,j] = 0$) results in an $O(s)$ operations algorithm. Thus, $\widetilde{O}(n)$-sparse matrices indeed satisfy the linear-time matrix-vector multiplication desiderata.

Next, we consider more algebraic families of matrices. Consider the *discrete Fourier matrix*:

Definition 2.4. *The $n \times n$* discrete Fourier *matrix $\mathbf{F}_n$ defined as follows (for $0 \leq i,j < n$):*

$$F_n[i,j] = \omega_n^{ij},$$

where $\omega_n = e^{-2\pi\iota/n}$ is the n-th root of unity *and $\iota = \sqrt{-1}$.*

We note that even though the discrete Fourier matrix has rank n and sparsity n^2, it has a very simple representation: just the number n.

Let us unroll the following matrix-vector multiplication: $\hat{\mathbf{x}} = \mathbf{F}_n \boldsymbol{x}$. In particular, for any $0 \leq i < n$:

$$\hat{\mathbf{x}}[i] = \sum_{j=0}^{n-1} \boldsymbol{x}[j] \cdot e^{2\pi\iota ji/n}.$$

In other words, $\hat{\mathbf{x}}$ is the *discrete Fourier transform* of $\boldsymbol{x}$. It turns out that the discrete Fourier transform is incredibly useful in practice (and is used in applications such as image compression). One of the most celebrated algorithmic results is that the Fourier transform can be computed with $O(n \log n)$ operations:

Theorem 2.5 (Fast Fourier Transform (FFT) [14])**.** *For any $\boldsymbol{x} \in \mathbb{C}^n$, one can compute $\mathbf{F}_n \cdot \boldsymbol{x}$ in $O(n \log n)$ operations.*

Thus, the discrete Fourier transform satisfies the near linear-time matrix-vector multiplication desiderata.

Consider the following matrix (called a Vandermonde matrix):

Definition 2.6 (Vandermonde Matrix)**.** *For any $n \geq 1$ and any field $\mathbb{F}$ with size at least m, m distinct elements $a_0, \ldots, a_{m-1} \in \mathbb{F}$, consider the matrix (where $0 \leq i < m$ and $0 \leq j < n$)*

$$\mathbf{V}_n^{(\boldsymbol{a})}[i,j] = a_i^j,$$

where $\boldsymbol{a} = (a_0, \ldots, a_{m-1})$.

We now state some interesting facts about these matrices (which also show that Vandermonde matrices satisfy the near linear-time matrix-vector multiplication desiderata):

1. One can represent a Vandermonde matrix by noting $a_0 \ldots, \alpha_{m-1}$ (along with n of course).
2. The discrete Fourier matrix is a special case of a Vandermonde matrix.
3. The Vandermonde matrix has full rank and has sparsity n^2.
4. It turns out that $\mathbf{V}_n \cdot \boldsymbol{x}$ for any $\boldsymbol{x} \in \mathbb{F}^n$ can be computed with $O(n \log^2 n)$ operations [11].

In all the four examples of structured matrices that we have seen in this section, their representation pretty much follows from their definitions. However, in general, whenever we have a *family* of *structured matrices*, we would like a generic way of referring to the representation. To abstract this we will assume that

> **Assumption 2.1.** *Given a vector $\boldsymbol{\theta} \in \mathbb{F}^s$ for some $s = s(m,n)$ such that the vector $\boldsymbol{\theta}$ completely specifies a matrix in our chosen family. We will use $\mathbf{W}_{\boldsymbol{\theta}}$ to denote the class of matrix family parameterized by $\boldsymbol{\theta}$.*

For example, if $s = mn$, then we get the set of all matrices in $\mathbb{F}^{m \times n}$. On the other hand, for say the Vandermonde matrix (recall Definition 2.6), we have $s(m,n) = m$ and $\boldsymbol{\theta} = (a_0, \ldots, a_{m-1})$ for distinct a_i's.

2.5 Arithmetic Circuits

So far we have tip-toed around how to determine the 'optimal' matrix vector multiplication time for a given $\mathbf{W}$. Now, we pay closer attention to this problem:

> **Question 2.1.** *Given an $m \times n$ matrix $\mathbf{W}$, what is the optimal complexity of computing $\mathbf{W} \cdot \boldsymbol{x}$ (for arbitrary $\boldsymbol{x}$)?*

Note that to even begin to answer the question above, we need to fix our 'machine model.' One natural model is the RAM model on which we analyze most of our beloved algorithms. However, we do not understand the power of RAM model (in the sense that we do not have a good handle on what problems can be solved by say linear-time or quadratic-time algorithms[7]) and answering Question 2.1 in the RAM model seems hopeless.

[7] The reader might have noticed that we are ignoring the P vs. NP elephant in the room.

So we need to consider a more restrictive model of computation. Instead of going through a list of possible models, we will just state the model of computation we will use: *arithmetic circuit* (also known as the straight-line program). In the context of an arithmetic circuit that computes $\boldsymbol{y} = \mathbf{W}\boldsymbol{x}$, there are n inputs gates (corresponding to $\boldsymbol{x}[0], \ldots, \boldsymbol{x}[n-1]$) and m output gates (corresponding to $\boldsymbol{y}[0], \ldots, \boldsymbol{y}[m-1]$). All the internal gates correspond to the addition, multiplication, subtraction and division operators over the underlying field $\mathbb{F}$. The circuit is also allowed to use constants from $\mathbb{F}$ for 'free.' The complexity of the circuit will be its *size*: i.e. the number of addition, multiplication, subtraction and division gates in the circuit. We will also care about the *depth* of the circuit, which is the depth of the DAG representing the circuit. Let us record this choice:

Definition 2.7. *For any function $f : \mathbb{F}^n \to \mathbb{F}^m$, its* arithmetic circuit complexity *is the minimum number of addition, multiplication, subtraction and division operations over $\mathbb{F}$ needed to compute $f(\boldsymbol{x})$ for any $\boldsymbol{x} \in \mathbb{F}^n$.*

Given the above, we have the following more specific version of Question 2.1:

Question 2.2. *Given a matrix $\mathbf{W} \in \mathbb{F}^{m \times n}$, what is the arithmetic circuit complexity of computing $\mathbf{W} \cdot \boldsymbol{x}$ (for arbitrary $\boldsymbol{x} \in \mathbb{F}^n$)?*

One drawback of arithmetic circuits (especially for infinite fields e.g. $\mathbb{F} = \mathbb{R}$, which is our preferred choice for deep learning applications) is that they assume operations over $\mathbb{F}$ can be performed *exactly*. In particular, it ignores precision issues involved with real arithmetic. Nonetheless, this model turns out to be a very useful model in reasoning about the complexity of doing matrix-vector multiplication for any family of matrices.

Perhaps the strongest argument in support of arithmetic circuits is that a large (if not an overwhelming) majority of matrix-vector multiplication algorithm in the RAM model also imply an arithmetic circuit of size comparable to the runtime of the algorithm (and the depth of the circuit roughly corresponds to the time taken to compute it by a parallel algorithm). For example consider the obvious algorithm to compute $\mathbf{W}\boldsymbol{x}$ (i.e. for each $i \in [m]$, compute $\boldsymbol{y}[i]$ as the sum $\sum_{i=0}^{n-1} \mathbf{W}[i,j]\boldsymbol{x}[j]$). It is easy to see that this algorithm implies an arithmetic circuit of size $O(nm)$ and depth $O(\log n)$.

One reason for the vast majority of existing efficient matrix vector algorithms leading to arithmetic circuits is that they generally are divide and conquer algorithms that use polynomial operations such as polynomial multiplication or evaluation (both of which themselves are divide and conquer algorithms that use FFT (Theorem 2.5) as a blackbox) or polynomial addition. Each of these pieces are well known to have small (depth and size) arithmetic circuits (since FFT has these properties). Finally, the divide and conquer structure of the algorithms leads to the circuit being of low depth. See the book of Pan [32] for a more elaborate description of this connection.

2.5.1 Linear circuit complexity

Next, instead of considering the general arithmetic circuit complexity of $\mathbf{W}\boldsymbol{x}$, let us consider the linear arithmetic circuit complexity. A linear arithmetic circuit only uses linear operations:

Definition 2.8. *A linear arithmetic circuit (over $\mathbb{F}$) only allows operations of the form $\alpha X + \beta Y$, where $\alpha, \beta \in \mathbb{F}$ are constants while X and Y are the inputs to the operation. The* linear arithmetic circuit complexity *of $\mathbf{W}\boldsymbol{x}$ is the size of the smallest linear arithmetic circuit that computes $\mathbf{W}\boldsymbol{x}$ (where $\boldsymbol{x}$ are the inputs and the circuit depends on $\mathbf{W}$). Sometimes we will overload terminology and call the (linear) arithmetic circuit complexity of computing $\mathbf{W}\boldsymbol{x}$ as the (linear) arithmetic circuit complexity of (just) $\mathbf{W}$.*

We first remark that the linear arithmetic circuit complexity seems to be a very natural model to consider the complexity of computing $\mathbf{W}\boldsymbol{x}$ (recall this defines a linear function over $\boldsymbol{x}$). In fact one could plausibly conjecture that going from general arithmetic circuit complexity to linear arithmetic circuit complexity of computing $\mathbf{W}\boldsymbol{x}$ should be without loss of generality (the intuition being: "What else can you do?").

It turns out that for infinite fields, the above intuition is correct:

Theorem 2.9 ([11])**.** *Let* $\mathbf{F}$ *be an infinite field and* $\mathbf{W} \in \mathbb{F}^{m\times n}$*. Let* $\mathscr{C}(\mathbf{W})$ *and* $\mathscr{C}^L(\mathbf{W})$ *be the arithmetic circuit complexity and linear arithmetic circuit complexity of computing* $\mathbf{W}\boldsymbol{x}$ *(for arbitrary* $\boldsymbol{x}$*). Then* $\mathscr{C}^L(\mathbf{W}) = \Theta(\mathscr{C}(\mathbf{W}))$.

We first make some observations. First, it turns out that Theorem 2.9 can be proved for finite fields that are exponentially large. Second, it is a natural question to try and prove a version of Theorem 2.9 for small finite fields (say over $\mathbb{F}_2$). This question is very much open.

2.6 Problem Definition

Finally, we have all the pieces in place so that we formally define the problem we are interested in.

Mainly for notational simplicity, we make the following assumption for the rest of the survey:

> **Assumption 2.2.** *Unless stated otherwise, we will consider square matrices, i.e.* $m = n$.

As mentioned in Section 2.3, we would like to use a weight matrix $\mathbf{W}$ such that computing $\mathbf{W}\boldsymbol{x}$ is efficient. In particular, using our choice of measuring algorithmic efficiency by the arithmetic complexity of computing $\mathbf{W}\boldsymbol{x}$, the design problem becomes the following– can we design neural networks with weight matrices $\mathbf{W}$ that are guaranteed to have an arithmetic circuit of size s (for some s that is at most $o(n^2)$)? In the rest of the section, we will successively formalize (and specialize) the above intuitive problem statement.

Recall from Section 2.3 that for neural networks, the main bottleneck is to be able to 'learn' these weight matrices $\mathbf{W}$ from the training data (we will formally state the learning problem in Definition 3.2 but for now we'll keep the definition of training a bit vague). But even before we talk about the efficiency[8] of learning the matrix $\mathbf{W}$, we note that it is important to be more precise of the representation that the learning algorithm outputs. In particular, even if $\mathbf{W}$ has an arithmetic circuit of size $s = o(n^2)$, if the learning algorithm outputs the matrix $\mathbf{W}$ in the usual $n \times n$ matrix format, then we are still stuck with an $\Omega(n^2)$ arithmetic circuit complexity for the learned matrix $\mathbf{W}$.

Thus, we want the learning process to not only learn a matrix $\mathbf{W}$ with arithmetic circuit complexity s but also to learn a representation from which one can easily create a matrix-vector multiplication algorithm with complexity (roughly) s. This implies that we first need to identify a *class* of structured matrices that can capture matrices with arithmetic circuit complexity of s. Allowing for the possibility that we might need more than s parameters to index the class of matrices we are after, here is a more formal version of the problem we had stated earlier:

- A parameter size $s' \geq s$ and a function $f : \mathbb{F}^{s'} \to \mathbb{F}^{n\times n}$ such that

[8] Recall that the training problem happens 'offline' so we do not need the learning to be say $O(n)$ time but we would like the learning algorithm to be at the worst be polynomial time.

1. For every matrix $\mathbf{W}$ with arithmetic circuit complexity at most s, there exists a $\boldsymbol{\theta} \in \mathbb{F}^{s'}$ such that $f(\boldsymbol{\theta}) = \mathbf{W}$.
2. Given $\boldsymbol{\theta}$ one can efficiently compute $f(\boldsymbol{\theta}) \cdot \boldsymbol{x}$ (here by efficiently we mean with roughly $\widetilde{O}(s')$ arithmetic operations).
3. We can efficiently learn the parameter $\boldsymbol{\theta}$ that defines $\mathbf{W}$.

- The overall goal would be to make s' as close to s as possible– ideally we want $s' = \widetilde{O}(s)$.

There is an 'obvious' family that almost gets us what we want– just define the parameter $\boldsymbol{\theta}$ to encode the circuit computing $\mathbf{W}\boldsymbol{x}$. The problem with this formulation (other than being not an 'interesting' definition) is that there is no known efficient way to learn the optimal arithmetic circuit for $\mathbf{W}$ (even if we were given access to the $n \times n$ representation of $\mathbf{W}$).

Another candidate for the class of circuits we are looking for will be the family of low rank matrices. In particular, given the target s, we would like to figure out the value of rank r so that we can pick $s' = rn$ and we use the standard representation of rank r matrices. In this case, it is easy to verify that all the three properties above are satisfied. The problem of learning the rank r decomposition of a given matrix $\mathbf{W}$ e.g. can be computed by the Singular Value Decomposition (or SVD).[9] Unfortunately, in general s' can be much larger than s– consider e.g. the DFT (Definition 2.4), which has $s = O(n \log n)$, but since the matrix is full rank, we need $r = n$ and hence $s' = n^2$, which is not that useful.

We will consider some other choices for families of structured matrices in Section 4 but before we finalize the problem statement, we use the following observation from practice to make the problem a bit more tractable– it turns out in practice that the weight matrix $\mathbf{W}$ (or its representation $\boldsymbol{\theta}$) is learned via gradient descent (see Algorithm 1). So we make the following assumption:

Assumption 2.3. *We will assume that we can only use gradient descent to learn the representation $\boldsymbol{\theta}$ for our target matrix $\mathbf{W}$.*

What the above means is that it is *sufficient* to be able to compute the gradient of f at any point in $\mathbb{F}^{s'}$ (see Section 3 for details on why this is the case). Under this assumption, we can modify our earlier goal into our final problem statement:

Question 2.3. *Does there exist a family of $n \times n$ matrices such that for every parameter $n \le s \le n^2$, there exists a parameter s' and a map $f : \mathbb{F}^{s'} \to \mathbb{F}^{n \times n}$ such that for every matrix $\mathbf{W}$ with arithmetic circuit complexity of at most s, there exists $\boldsymbol{\theta} \in \mathbb{F}^{s'}$ such that $f(\boldsymbol{\theta}) = \mathbf{W}$. Furthermore, we want*

- *(*Expressivity property*) s' is as close to s as possible (ideally $s' = \widetilde{O}(s)$)*
- *(*Efficient MVM property*) Given $\boldsymbol{\theta}$, we can compute $f(\boldsymbol{\theta}) \cdot \boldsymbol{x}$ for any $\boldsymbol{x} \in \mathbb{F}^n$ in close to s' arithmetic operations.*
- *(*Efficient gradient property*) For any $\boldsymbol{a} \in \mathbb{F}^{s'}$, one can evaluate the gradient of f at $\boldsymbol{a}$ efficiently (ideally as close to s' arithmetic operations as possible).*

[9] In fact the SVD will give the best rank r approximation even if $\mathbf{W}$ is not rank r– for now let's just consider the problem setting where we are looking for an *exact* representation.

Before we attack Question 2.3, we will take a bit of a detour to consider the problem of learning $\mathbf{W}$ from training data in more detail.

3 Computing gradients

We will formalize the problem of learning from training data in Section 3.1. Then in Section 3.2, we identify a specific gradient function that is sufficient to run gradient descent for our purposes. We recall the Baur-Strassen theorem in Section 3.3, which will show that for our gradient problem, it is enough to ensure that $\mathbf{W}$ has small arithmetic circuit complexity. Finally, in Section 3.4, we take a detour to highlight a really cool result, which unfortunately does not seem to be as well-known as it should be.

We will *not* be assuming Assumption 2.2 in this section, i.e. in this section we will consider a general rectangular matrix $\mathbf{W}$ (and we will revert to Assumption 2.2 from next section onwards).

3.1 Back to (not so) deep learning

We go back to the single layer neural network that we studied earlier in Section 2.3. In particular, recall we consider a single layer neural network that is defined by

$$\boldsymbol{y} = g\left(\mathbf{W} \cdot \boldsymbol{x}\right), \tag{1}$$

where $\mathbf{W} \in \mathbb{F}^{m \times n}$ and $g : \mathbb{F}^m \to \mathbb{F}^m$ is a non-linear function. Further,

Assumption 3.1. *We will assume that non-linear function $g : \mathbb{F}^m \to \mathbb{F}^m$ is obtained by applying the same function $g : \mathbb{F} \to \mathbb{F}$ to each of the m elements.*

In other words, equation 1 is equivalently stated as for every $0 \leq i < m$:

$$\boldsymbol{y}[i] = g\left(\langle \mathbf{W}[i,:], \boldsymbol{x} \rangle\right).$$

Recall that in Section 2.3, we had claimed (without any argument) that the complexity of learning the weight matrix $\mathbf{W}$ given few samples is governed by the complexity of matrix-vector multiplication for $\mathbf{W}$. In this section, we will rigorously argue this claim. To do this, we define the learning problem more formally:

Definition 3.1. *Given L* training data $\left(\boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)}\right)$ *for $\ell \in [L]$, we want to compute a matrix $\mathbf{W} \in \mathbb{F}^{m \times n}$ that minimizes the error*

$$E(\mathbf{W}) = \sum_{\ell=1}^{L} \left\| \boldsymbol{y}^{(\ell)} - g\left(\mathbf{W} \cdot \boldsymbol{x}^{(\ell)}\right) \right\|_2^2.$$

We note that the above is not the only *error function* that is used in training neural networks but the above is a common choice and hence, we stick with it. Further, note that in the above the training searches for the 'best' weight matrix from the set of all matrices in $\mathbb{F}^{m \times n}$. However, since we are interested in searching for the best weight matrix with a certain class as in Question 2.3, we generalize Definition 3.1 as follows:

Definition 3.2. *Given L* training data $\left(\boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)}\right)$ *for $\ell \in [L]$, we want to compute the parameters of an $m \times n$ matrix $\boldsymbol{\theta} \in \mathbb{F}^{s(m,n)}$ that minimizes the error (where we use $\mathbf{W}_{\boldsymbol{\theta}} = f(\boldsymbol{\theta})$):*

$$E(\boldsymbol{\theta}) = \sum_{\ell=1}^{L} \left\| \boldsymbol{y}^{(\ell)} - g\left(\mathbf{W}_{\boldsymbol{\theta}} \cdot \boldsymbol{x}^{(\ell)}\right) \right\|_2^2.$$

3.1.1 Gradients and Gradient Descent

(Partial) Derivatives. It turns out that we will only be concerned with studying derivatives of polynomials. For this, we can define the notion of a formal derivative (over univariate polynomials):

Definition 3.3. *The formal derivative* $\nabla_X(\cdot) : \mathbb{F}[X] \to \mathbb{F}[X]$ *is defined as follows. For every integer* i*,*

$$\nabla_X\left(X^i\right) = i \cdot X^{i-1}.$$

The above definition can be extended to all polynomials in $\mathbb{F}[X]$ *by insisting that* $\nabla_X(\cdot)$ *be a linear map. That is for every* $\alpha, \beta \in \mathbb{F}$ *and* $f(X), g(X) \in \mathbb{F}[X]$ *we have*

$$\nabla_X\left(\alpha f(X) + \beta g(X)\right) = \alpha \nabla_X\left(f(X)\right) + \beta \nabla_X\left(g(X)\right).$$

We note that over $\mathbb{R}$, the above definition when applied to polynomials over $\mathbb{R}[X]$ gives the same result as the usual notion of derivatives.

We will actually need to work with derivatives of multi-variate polynomials. We will use $\mathbb{F}[X_1, \ldots, X_m]$ to denote the set of multivariate polynomials with variables $X_1, \ldots, X_m$. For example, $3XY + Y^2 + 1.5X^3Y^4$ is in $\mathbb{R}[X, Y]$. We extend the definition of derivatives from Definition 3.3 to the following (which also called a *gradient*)

Definition 3.4. *Let* $f(X_1, \ldots, X_n)$ *be a polynomial in* $\mathbb{F}[X_1, \ldots, X_n]$*. Then define its* gradient *as (where we use* $\mathbf{X} = (X_1, \ldots, X_n)$ *to denote the vector of variables):*

$$\nabla_{\mathbf{X}}\left(f(\mathbf{X})\right) = \left(\nabla_{X_1}\left(f(\mathbf{X})\right), \ldots, \nabla_{X_n}\left(f(\mathbf{X})\right)\right),$$

where in $\nabla_{X_i}\left(f(\mathbf{X})\right)$*, we think of* $f(\mathbf{X})$ *as being a polynomial in* X_i *with coefficients in* $\mathbb{F}[X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n]$*.*

Finally note that $\nabla_{X_i}\left(f(\mathbf{X})\right)$ *is again a polynomial and we will denote its evaluation at* $\boldsymbol{a} \in \mathbb{F}^n$ *as* $\nabla_{X_i}\left(f(\mathbf{X})\right)_{|\boldsymbol{a}}$*. We extend this notation to the gradient by*

$$\nabla_{\mathbf{X}}\left(f(\mathbf{X})\right)_{|\boldsymbol{a}} = \left(\nabla_{X_1}\left(f(\mathbf{X})\right)_{|\boldsymbol{a}}, \ldots, \nabla_{X_n}\left(f(\mathbf{X})\right)_{|\boldsymbol{a}}\right).$$

For example

$$\nabla_{X,Y}\left(3XY + Y^2 + 1.5X^3Y^4\right) = \left(3Y + 4.5X^2Y^4, 3X + 2Y + 6X^3Y^3\right).$$

Gradient Descent. While there exist techniques to solve the above problem theoretically, in practice *Gradient Descent* is commonly used to solve the above problem. In particular, one starts off with an initial state $\boldsymbol{\theta} = \boldsymbol{\theta}_0 \in \mathbb{F}^s$ and one keeps changing $\boldsymbol{\theta}$ is opposite direction of $\nabla_{\boldsymbol{\theta}}(E(\boldsymbol{\theta}))$ till the error is below a pre-specified threshold (or one goes beyond a pre-specified number of iterations). Algorithm 1 has the details.

3.2 Computing the gradient

It is clear from Algorithm 1, that the most computationally intensive part is computing the gradient. We first show that if one can compute a related gradient, then we could implement Algorithm 1. In Section 3.3 we will show that this latter gradient computation is closely tied to computing $\mathbf{W}\boldsymbol{x}$. We first argue:

Algorithm 1 Gradient Descent

INPUT: $\eta > 0$ and $\varepsilon > 0$
OUTPUT: $\boldsymbol{\theta}$

$i \leftarrow 0$
Pick $\boldsymbol{\theta}_0$ ▷ This could be arbitrary or initialized to something more specific
WHILE $|E(\boldsymbol{\theta}_i)| \geq \varepsilon$ DO ▷ One could also terminate based on number of iterations
    $\boldsymbol{\theta}_{i+1} \leftarrow \boldsymbol{\theta}_i - \eta \cdot (\nabla_{\boldsymbol{\theta}} (E(\boldsymbol{\theta})))_{|\boldsymbol{\theta}_i}$ ▷ η is the 'learning rate'
    $i \leftarrow i + 1$
RETURN $\boldsymbol{\theta}_i$

Lemma 3.5. *If for every $\boldsymbol{z} \in \mathbb{F}^m$ and $\boldsymbol{u} \in \mathbb{F}^n$, one can compute $\left(\nabla_{\boldsymbol{\theta}} \left(\boldsymbol{z}^T \mathbf{W}_{\boldsymbol{\theta}} \boldsymbol{u}\right)\right)_{|\boldsymbol{a}}$ for any $\boldsymbol{a} \in \mathbb{F}^s$ in $T_1(m,n)$ operations and $\mathbf{W}\boldsymbol{u}$ in $T_2(m,n)$ operations, then one can compute $(\nabla_{\boldsymbol{\theta}} (E(\boldsymbol{\theta})))_{|\boldsymbol{\theta}_0}$ for a fixed $\boldsymbol{\theta}_0 \in \mathbb{F}^s$ in $O(L(T_1(m,n) + T_2(m,n)))$ operations.*

Proof. For notational simplicity define

$$\mathbf{W} = \mathbf{W}_{\boldsymbol{\theta}_0}$$

and

$$E_\ell(\boldsymbol{\theta}) = \left\| \boldsymbol{y}^{(\ell)} - g\left(\mathbf{W}_{\boldsymbol{\theta}} \cdot \boldsymbol{x}^{(\ell)}\right) \right\|_2^2.$$

Fix $\ell \in [L]$. We will show that we can compute $\nabla_{\boldsymbol{\theta}} (E_\ell(\boldsymbol{\theta}))_{|\boldsymbol{\theta}_0}$ with $O(T_1(m,n) + T_2(m,n))$ operations, which would be enough since $\nabla_{\boldsymbol{\theta}} (E(\boldsymbol{\theta})) = \sum_{\ell=1}^{L} \nabla_{\boldsymbol{\theta}} (E_\ell(\boldsymbol{\theta}))$.

For notational simplicity, we will use $\boldsymbol{y}, \boldsymbol{x}$ and $E(\boldsymbol{\theta})$ to denote $\boldsymbol{y}^{(\ell)}, \boldsymbol{x}^{(\ell)}$ and $E_\ell(\boldsymbol{\theta})$ respectively. Note that

$$\begin{aligned} E(\boldsymbol{\theta}) &= \left\| \boldsymbol{y} - g\left(\mathbf{W}_{\boldsymbol{\theta}} \cdot \boldsymbol{x}\right) \right\|_2^2 \\ &= \sum_{i=0}^{m-1} \left(\boldsymbol{y}[i] - g\left(\sum_{j=0}^{n-1} \mathbf{W}_{\boldsymbol{\theta}}[i,j] \boldsymbol{x}[j] \right) \right)^2. \end{aligned}$$

Applying the chain rule of the gradient on the above, we get (where $g'(x)$ is the derivative of $g(x)$):

$$\nabla_{\boldsymbol{\theta}} (E(\boldsymbol{\theta})) = -2 \sum_{i=0}^{m-1} \left(\boldsymbol{y}[i] - g\left(\sum_{j=0}^{n-1} \mathbf{W}_{\boldsymbol{\theta}}[i,j] \boldsymbol{x}[j] \right) \right) g'\left(\sum_{j=0}^{n-1} \mathbf{W}_{\boldsymbol{\theta}}[i,j] \boldsymbol{x}[j] \right) \sum_{j=1}^{n-1} \left(\nabla_{\boldsymbol{\theta}} \left(\mathbf{W}_{\boldsymbol{\theta}}[i,j] \right) \boldsymbol{x}[j] \right). \tag{2}$$

Define a vector $\boldsymbol{z} \in \mathbb{F}^m$ such that for any $0 \leq i < m$,

$$\boldsymbol{z}[i] = -2 \left(\boldsymbol{y}[i] - g\left(\langle \mathbf{W}[i,:], \boldsymbol{x} \rangle \right) \right) g'\left(\langle \mathbf{W}[i,:], \boldsymbol{x} \rangle \right).$$

Note that once we compute $\mathbf{W}\boldsymbol{x}$ (which by assumption we can do in $T_2(m,n)$ operation), we can compute $\boldsymbol{z}$ with $O(T_2(m,n))$ operations.[10] Further, note that $\boldsymbol{z}$ is independent of $\boldsymbol{\theta}$ (recall $\mathbf{W} = \mathbf{W}_{\boldsymbol{\theta}_0}$).

[10] Here we have assumed that one can compute $g(x)$ and $g'(x)$ with $O(1)$ operations and assumed that $T_2(m,n) \geq m$.

From (2), we get that

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}}\left(E(\boldsymbol{\theta})\right)_{|\boldsymbol{\theta}_0} &= -2\sum_{i=0}^{m-1}\left(\boldsymbol{y}[i]-g\left(\langle \mathbf{W}[i,:],\boldsymbol{x}\rangle\right)\right)g'\left(\langle \mathbf{W}[i,:],\boldsymbol{x}\rangle\right)\sum_{j=0}^{n-1}\left(\nabla_{\boldsymbol{\theta}}\left(\mathbf{W}_{\boldsymbol{\theta}}[i,j]\right)_{|\boldsymbol{\theta}_0}\cdot\boldsymbol{x}[j]\right)\\
&= \sum_{i=0}^{m-1}\boldsymbol{z}[i]\cdot\sum_{j=0}^{n-1}\left(\nabla_{\boldsymbol{\theta}}\left(\mathbf{W}_{\boldsymbol{\theta}}[i,j]\right)_{|\boldsymbol{\theta}_0}\cdot\boldsymbol{x}[j]\right)\\
&= \left(\nabla_{\boldsymbol{\theta}}\left(\sum_{i=0}^{m-1}\boldsymbol{z}[i]\cdot\sum_{j=0}^{n-1}\mathbf{W}_{\boldsymbol{\theta}}[i,j]\cdot\boldsymbol{x}[j]\right)\right)_{|\boldsymbol{\theta}_0}\\
&= \left(\nabla_{\boldsymbol{\theta}}\left(\boldsymbol{z}^T\mathbf{W}_{\boldsymbol{\theta}}\boldsymbol{x}\right)\right)_{|\boldsymbol{\theta}_0}.
\end{aligned}$$

In the above, the first equality follows from our notation that $\mathbf{W}=\mathbf{W}_{\boldsymbol{\theta}_0}$, the second equality follows from the definition of $\boldsymbol{z}$ and the third equality follows from the fact that $\boldsymbol{z}$ is independent of $\boldsymbol{\theta}$. The proof is complete by noting that we can compute $\left(\nabla_{\boldsymbol{\theta}}\left(\boldsymbol{z}^T\mathbf{W}_{\boldsymbol{\theta}}\boldsymbol{x}\right)\right)_{|\boldsymbol{\theta}_0}$ in $T_1(m,n)$ operations. □

Thus, to efficiently implement gradient descent, we have to efficiently compute $\left(\nabla_{\boldsymbol{\theta}}\left(\boldsymbol{z}^T\mathbf{W}_{\boldsymbol{\theta}}\boldsymbol{x}\right)\right)_{|\boldsymbol{\theta}_0}$ for any fixed $\boldsymbol{z}\in\mathbb{F}^m$ and $\boldsymbol{x}\in\mathbb{F}^n$. Next, we will show that the arithmetic complexity of this operation is the same (up to constant factors) as the arithmetic complexity of computing $\boldsymbol{z}^T\mathbf{W}\boldsymbol{x}$ (which in turn has complexity no worse than that of computing our old friend $\mathbf{W}\boldsymbol{x}$). In the next section, not only will we show that this result is true but it is true for *any* function $f:\mathbb{F}^s\to\mathbb{F}$. As a bonus, we will present a simple (but somewhat non-obvious) algorithmic proof.

3.3 Computing gradients very fast

In this section we consider the following general problem:

- **Input**: An arithmetic circuit $\mathscr{C}$ that computes a function $f:\mathbb{F}^s\to\mathbb{F}$ and an evaluation point $\boldsymbol{a}\in\mathbb{F}^s$.
- **Output**: $\nabla_{\boldsymbol{\theta}}\left(f(\boldsymbol{\theta})\right)_{|\boldsymbol{a}}$.

Recall that in the previous section, we were interested in solving the above problem for the function $f_{\boldsymbol{z},\boldsymbol{x}}(\boldsymbol{\theta})=\boldsymbol{z}^T\mathbf{W}_{\boldsymbol{\theta}}\boldsymbol{x}$ where $\mathbf{W}_{\boldsymbol{\theta}}\in\mathbb{F}^{m\times n}$, $\boldsymbol{z}\in\mathbb{F}^m$ and $\boldsymbol{x}\in\mathbb{F}^n$.

The way we will tackle the above problem is given the arithmetic circuit $\mathscr{C}$ for $f(\boldsymbol{\theta})$, we will try to come up with an arithmetic circuit $\mathscr{C}'$ to compute $\nabla_{\boldsymbol{\theta}}\left(f(\boldsymbol{\theta})\right)$. We first note that given a fixed $0\le\ell<s$, it is fairly easy compute a circuit $\mathscr{C}'_\ell$ that on input $\boldsymbol{a}\in\mathbb{F}^s$ computes $\nabla_{\boldsymbol{\theta}[\ell]}\left(f(\boldsymbol{\theta})\right)_{|\boldsymbol{a}}$ with essentially the same size. This implies that one can compute $\nabla_{\boldsymbol{\theta}}\left(f(\boldsymbol{\theta})\right)$ with arithmetic circuit complexity $O(m\cdot|\mathscr{C}|)$ (where $|\mathscr{C}|$ denotes the size of $\mathscr{C}$).

We will now recall the Baur-Strassen theorem, which states that the gradient can be computed in the same (up to constant factors) arithmetic circuit complexity as evaluating f.

Theorem 3.6 (Baur-Strassen Theorem [6])**.** *Let $f:\mathbb{F}^s\to\mathbb{F}$ be a function that has an arithmetic circuit $\mathscr{C}$ such that given $\boldsymbol{\theta}\in\mathbb{F}^s$, it computes $f(\boldsymbol{\theta})$. Then there exists another arithmetic circuit $\mathscr{C}'$ that computes for any given $\boldsymbol{a}\in\mathbb{F}^s$, the gradient $\nabla_{\boldsymbol{\theta}}\left(f(\boldsymbol{\theta})\right)_{|\boldsymbol{a}}$. Further,*

$$|\mathscr{C}'|\le O(|\mathscr{C}|).$$

The proof of Baur-Strassen theorem is actually algorithmic– Algorithm 2 shows how to compute the gradient given the arithmetic circuit for f (it is not too hard to see that the algorithm implicitly defines the claimed arithmetic circuit $\mathscr{C}'$). The proof of correctness of the algorithm follows from the following version of chain rule for multi-variable function.

Lemma 3.7. *Let $f : \mathbb{F}^s \to \mathbb{F}$ be a function composition of a polynomial $g \in \mathbb{F}[H_1, \ldots, H_k]$ and polynomials $h_i \in \mathbb{F}[X_1, \ldots, X_s]$ for every $i \in [k]$, i.e.*

$$f(\mathbf{X}) = g\left(h_1(\mathbf{X}), \ldots, h_k(\mathbf{X})\right).$$

Then for every $0 \leq \ell < s$, we have

$$\nabla_{X_\ell}\left(f(\mathbf{X})\right) = \sum_{j=1}^{k} \nabla_{H_j}\left(g(H_1, \ldots, H_k)\right) \cdot \nabla_{X_\ell}\left(h_j(\mathbf{X})\right).$$

We note that over $\mathbb{R}$ the above is known as the *high-dimensional chain rule* (and it holds for more general classes of functions). It turns out that if g and h_i are polynomials, then the high-dimensional chain rule pretty much follows from Definition 3.3.

Algorithm 2 Back-propagation Algorithm

INPUT: $\mathscr{C}$ that computes a function $f : \mathbb{F}^s \to \mathbb{F}$ and an evaluation point $\boldsymbol{a} \in \mathbb{F}^s$
OUTPUT: $\nabla_{\boldsymbol{\theta}}\left(f(\boldsymbol{\theta})\right)_{|\boldsymbol{a}}$

Let σ be an ordering of gates of $\mathscr{C}$ in reverse topological sort with output gate first ▷ This is possible since the graph of $\mathscr{C}$ is a DAG
WHILE Next gate g in σ has not been considered DO
    Let the parent gates of g be $h_1, \ldots, h_k$ ▷ $k = 0$ is allowed and implies no parents
    IF $k = 0$ THEN
        $\boldsymbol{d}[g] \leftarrow 1$
    ELSE
        $\boldsymbol{d}[g] \leftarrow 0$
        FOR $i \in [k]$ DO
            $\boldsymbol{d}[g] \leftarrow \boldsymbol{d}[g] + \nabla_g\left(h_i\right)_{|\boldsymbol{a}} \cdot \boldsymbol{d}[h_i]$
RETURN $(\boldsymbol{d}[\theta_i])_{0 \leq i < s}$ ▷ $\theta_0, \ldots, \theta_{s-1}$ are input gates

Theorem 3.6 and Lemma 3.5 imply the following connection between the gradient we want to compute the arithmetic circuit complexity of the corresponding matrix-vector multiplication problem:

Corollary 3.8. *If for every $\boldsymbol{\theta} \in \mathbb{F}^s$, $\mathbf{W}_{\boldsymbol{\theta}}$ has arithmetic circuit complexity of m, then we can compute $\left(\nabla_{\boldsymbol{\theta}}\left(E(\boldsymbol{\theta})\right)\right)_{|\boldsymbol{\theta}_0}$ for every $\boldsymbol{\theta}_0 \in \mathbb{F}^s$ in $O(L(m+n))$ operations.*

3.3.1 Automatic Differentiation

It turns out that Algorithm 2 can be extended to work beyond arithmetic circuits (at least over $\mathbb{R}$). This uses that fact that the high dimensional chain rule (Lemma 3.7) holds for any differentiable functions $g, h_1, \ldots, h_k$. In other words, we can consider circuits that compute f where each gate computes a differentiable function of its input. In other words, given a circuit for f with 'reasonable' gates, one can

automatically compile another circuit for its gradient. This idea has lead to the creation of the field of *automatic differentiation* (or *auto diff*) and is at the heart of many recent machine learning progress. In particular, those familiar with neural networks would notice that Algorithm 2 is the well-known *back-propagation algorithm* (and hence the title of Algorithm 2). However, for this survey, we will not need the full power of auto diff (Corollary 3.8 is all we need).

Next, we take a (wide) detour and state a result that is not as well-known as it should be.

3.4 Multiplying by the transpose

We first recall the definition of the transpose of a matrix:

Definition 3.9. *The* transpose *of a matrix* $\mathbf{A} \in \mathbb{F}^{m \times n}$*, denoted by* $\mathbf{A}^T \in \mathbb{F}^{n \times m}$ *is defined as follows (for any* $0 \le i < n, 0 \le j < m$*:*

$$\mathbf{A}^T[i,j] = \mathbf{A}[j,i].$$

It is natural to ask (since the transpose it so closely related to the original matrix):

> **Question 3.1.** *Is the (arithmetic circuit) complexity of computing* $\mathbf{A}^T \boldsymbol{x}$ *related to the (arithmetic circuit) complexity of computing* $\mathbf{A}\boldsymbol{x}$ *for* every *matrix* $\mathbf{A} \in \mathbb{F}^{m \times n}$*? E.g. are they within* $\widetilde{O}(1)$ *of each other?*

We will address the above question in the rest of this section.

3.4.1 Transposition principle

It turns out that the answer to Question 3.1 is an emphatic *yes*:

Theorem 3.10 (Transposition Principle [20])**.** *Fix a matrix* $\mathbf{A} \in \mathbb{F}^{n \times n}$ *such that there exists an arithmetic circuit of size* s *that computes* $\mathbf{A}\boldsymbol{x}$ *for arbitrary* $\boldsymbol{x} \in \mathbb{F}^n$*. Then there exists an arithmetic circuit of size* $O(s+n)$ *that computes* $\mathbf{A}^T \boldsymbol{y}$ *for arbitrary* $\boldsymbol{y} \in \mathbb{F}^n$*.*

> The above result was surprising to the author when he first came to know about it. Indeed, the knowledge of this result would have saved the author more than a year's worth of plodding while working on the paper [18]. For whatever reason, this result is not as well-known.

It is not too hard to show that the additive n term in the bound in the transposition principle is necessary.

There exist proofs of the transposition principle that are very structural in the sense that they consider the circuit for computing $\mathbf{A}\boldsymbol{x}$ and then directly change it to compute a circuit for $\mathbf{A}^T \boldsymbol{y}$.[11] For this survey we will present a much slicker proof that directly uses the Baur-Strassen theorem (to the best of our knowledge this proof was first explicitly stated in [27]). For this the following alternate view of $\mathbf{A}^T \boldsymbol{y}$ will be very useful:

$$\boldsymbol{y}^T \mathbf{A} = \left(\mathbf{A}^T \boldsymbol{y}\right)^T. \tag{3}$$

[11] At a very high level this involves 'reversing' the direction of the edges in the DAG corresponding to the circuit.

Proof of Theorem 3.10. Thanks to (3), we will consider the computation of $\boldsymbol{y}^T\mathbf{A}$ for any $\boldsymbol{y} \in \mathbb{F}^n$. We first claim that:

$$\boldsymbol{y}^T\mathbf{A} = \nabla_{\boldsymbol{x}}\left(\boldsymbol{y}^T\mathbf{A}\boldsymbol{x}\right). \tag{4}$$

Note that the function $\boldsymbol{y}^T\mathbf{A}\boldsymbol{x}$ is exactly the same product we have encountered before in Lemma 3.5.[12] Then note that given an arithmetic circuit of size s to compute $\mathbf{A}\boldsymbol{x}$ one can design an arithmetic circuit that computes $\boldsymbol{y}^T\mathbf{A}\boldsymbol{x}$ of size $s + O(n)$ (by simply additionally computing $\langle \boldsymbol{y}, \mathbf{A}\boldsymbol{x}\rangle$, which takes $O(n)$ operations.).

Now, by Theorem 3.6, there is a circuit that computes $\nabla_{\boldsymbol{x}}\left(\boldsymbol{y}^T\mathbf{A}\boldsymbol{x}\right)$ with arithmetic circuit of size $O(s+n)$.[13] Equation (4) completes the proof. □

4 Towards answering Question 2.3

In this section, we walk through some well studied classes of structured matrices and see how they all fall short of answering Question 2.3 fully.

4.1 Low rank matrices

We start with low rank matrices: we already addressed why low rank matrices cannot be the answer for Question 2.3 in Section 2.6 but we'll walk through the three requirements again. We consider the standard representation of a rank r matrix $\mathbf{W}$ as $\mathbf{W} = \mathbf{L}\cdot\mathbf{R}$ for $\mathbf{L} \in \mathbb{F}^{n\times r}$ and $\mathbf{R} \in \mathbb{F}^{r\times n}$. In this case $s' = 2rn$ and $\boldsymbol{\theta}$ is just the listing of all the entries in $\mathbf{L}$ and $\mathbf{R}$ and f is defined in the obvious way.

1. (Expressivity property) We have $s' = 2rn$. Consider the case e.g. when $\mathbf{W}$ is the discrete Fourier matrix, which has rank $r = n$ (and hence $s' \geq \Omega(n^2)$) and by Theorem 2.5, $\mathbf{W}$ has $s = O(n\log n)$. Thus, Expressivity property is not satisfied since the gap between s' and s is pretty much as large as possible.

2. (Efficient MVM property) This property is satisfied since the obvious matrix-vector multiplication algorithm (given $\mathbf{L}$ and $\mathbf{R}$) takes $O(rn)$ operations.

3. (Efficient gradient property) It is easy to see that each entry in $\mathbf{L}\cdot\mathbf{R}$ is a degree two polynomial in the entries of $\boldsymbol{\theta}$ and hence is also differentiable.

4.2 Sparse matrices (in listing representation)

Next, we consider m sparse matrices in *listing representation*. In other words, $s' = O(m)$ and $\boldsymbol{\theta}$ is basically a list of triples (x_i, y_i, c_i) for $1 \leq i \leq m$. The map f is defined as follows:

$$f(\boldsymbol{\theta})[j,k] = \begin{cases} c & \text{if } (j,k,c) \text{ is in } \boldsymbol{\theta} \\ 0 & \text{otherwise} \end{cases}.$$

It turns out that sparse matrices do not satisfy two of the three requirements in Question 2.3–

[12] However, earlier we where taking the gradient with respect to (essentially) $\mathbf{A}$ whereas here it is with respect to $\boldsymbol{x}$.

[13] Here we consider $\mathbf{A}$ as given and $\boldsymbol{x}$ and $\boldsymbol{y}$ as inputs. This implies that we need to prove the Baur-Strassen theorem when we only take derivatives with respect to part of the inputs– but this follows trivially since one can just read off $\nabla_{\boldsymbol{x}}\left(\boldsymbol{y}^T\mathbf{A}\boldsymbol{x}\right)$ from $\nabla_{\boldsymbol{x},\boldsymbol{y}}\left(\boldsymbol{y}^T\mathbf{A}\boldsymbol{x}\right)$.

- (EXPRESSIVITY PROPERTY) We have $s' = \Theta(m)$. However, for the discrete Fourier transform we have $m = n^2$ and as we have already observed that for the discrete Fourier transform we have $s = O(n \log n)$. Hence, the gap between s' and s is as large as possible.

- (EFFICIENT MVM PROPERTY) The obvious algorithm to multiply an m-sparse matrix with an arbitrary vector takes $O(m)$ operations and hence EFFICIENT MVM PROPERTY is satisfied.

- (EFFICIENT GRADIENT PROPERTY) It is easy to check that f as defined above is not differentiable (because the locations of the non-zero values are discrete). E.g. consider the case of $m = 1$ and let (x, y) be the location of the non-zero value (and let us assume that $\mathbf{W}[x, y] = 1$). In this case $f(\boldsymbol{\theta})[j, k] = \delta_{x=j, y=k}$, where δ is the Kronecker delta function for which the derivative is not defined at the point $(x, y, 1)$ and hence f is not differentiable.[14]

As bit of a spoiler alert, (variants) of sparse matrices will actually be crucial in answering Question 2.3 in the affirmative. It turns out that to satisfy EXPRESSIVITY PROPERTY one needs to consider *product* of sparse matrices (see Section 4.6) and to satisfy EFFICIENT GRADIENT PROPERTY one needs to go beyond the listing representation (see Section 5).

4.3 Sparse+low rank

Next, we consider the combination of sparse and low rank matrices. Not only is this a natural combination to consider but such matrices have been well-studied in the context of *robust PCA* [12]. However, for this survey we are interested in this family of matrices since this is *exactly* the class of matrices considered in the *matrix rigidity* problem introduced by Valiant[41]. In particular, we recall the following result due to Valiant (where the specific statement is from Paturi and Pudlák [34]):

Theorem 4.1 ([41, 34])**.** *Let r, d, σ be positive integers such that $d > 4\log_2 \sigma$. Assume $\mathbf{W}$ has a circuit C with size*

$$s \leq r \cdot \frac{\log_2 d}{2\log_2\left(\frac{d}{4\log_2 \sigma}\right)},$$

and depth d. Then we can decompose $\mathbf{W}$ as

$$\mathbf{W} = \mathbf{S} + \mathbf{LR},$$

where both $\mathbf{S} \in \mathbb{F}^{m \times n}$ and $\mathbf{R} \in \mathbb{F}^{r \times n}$ are σ- row sparse (i.e. overall they are $m\sigma$ and $r\sigma$ sparse respectively) and $\mathbf{L} \in \mathbb{F}^{n \times r}$. In other words, $\mathbf{W}$ can be written as a sum of rank r and σn-sparse matrix.

The above result has spawned a long line of beautiful work in the area of matrix rigidity, which we do not have the space to do any justice, see the course notes by Golovnev [22] for more details.

Unfortunately, sparse+low-rank matrices cannot answer Question 2.3 positively either:

- (EXPRESSIVITY PROPERTY) It turns out that the discrete Fourier transform still shows that this property is not satisfied for sparse+low rank matrices though the gap between s' and s is not as dramatic

[14] In this survey we are dealing with the classical definition of derivatives. If one defines the Kronecker delta function as a limit of a distribution and consider derivatives in the sense of theory of distributions then EFFICIENT GRADIENT PROPERTY will be satisfied. Indeed, many practical implementation that use sparse as the weight matrices $\mathbf{W}$, when trying to learn $\mathbf{W}$ use the distributional definition of the Kronecker delta function.

as before. First by the result of Dvir and Liu [19], one can indeed get an rank $r + \sigma n$-sparse decomposition of the discrete Fourier transform such that $s' = 2rn + \sigma n$ is $o(n^2)$. Unfortunately for our purposes [19] is only able to show that $s' = n^{2-o(1)}$.[15] To the best of our knowledge the best parameters we can get in our setup is from Theorem 4.1. However, an application of Theorem 4.1 overall settings of parameters will still lead to $s' \geq \Omega(n^{5/4-\varepsilon})$ for any $\varepsilon > 0$.[16] Thus, while the gap is not quadratic as it was for the sparse only or low-rank only case, the gap is still too large for what we are after.

- (EFFICIENT MVM PROPERTY) Since this property is satisfied for rank r and σn-sparse matrices, this property is also satisfied for their sum.

- (EFFICIENT GRADIENT PROPERTY) Since this property is not satisfied for sparse matrices (with the listing representation), this property is not satisfied for sum of low rank and sparse matrices as well.

We would like to stress that the goal of matrix rigidity is different from ours in that the goal of the program of matrix rigidity is to exhibit an explicit matrix for which any decomposition as $\mathbf{R} + \mathbf{S}$ for $\mathbf{R}$ being rank $O\left(\frac{n}{\log\log n}\right)$ needs $\mathbf{S}$ to have sparsity $\Omega\left(n^{1+\varepsilon}\right)$ for some constant $\varepsilon > 0$. In our context we would have liked to show that matrices with small arithmetic circuits are *not* rigid.

4.4 Vandermonde matrices

So far we have been able to rule out low rank, sparse and sparse+low rank matrices just based on the discrete Fourier transform. However, the discrete Fourier transform by itself does not need a lot of parameters. In particular, it is a special case of Vandermonde matrices (Definition 2.6). It is natural to consider Vandermonde matrices as a potential answer to Question 2.3. In this case we use the obvious representation where $\boldsymbol{\theta}$ is just the vector $(a_1, \ldots, a_n)$ and f is defined as per Definition 2.6. Unfortunately, Vandermonde matrices cannot answer Question 2.3 positively either:

1. (EXPRESSIVITY PROPERTY) We have $s' = O(n \log^2 n)$ [11]. However, by a simple counting argument it is easy to see that Vandermonde matrices cannot represent all matrices. Specifically, consider the set of $\bar{s}$-sparse matrices with sparsity $\bar{s} = \omega(n)$. Since a Vandermonde matrix is represented by n parameters, there will be at least one $\bar{s}$-sparse matrix that cannot be represented as a Vandermonde matrix. Thus, EXPRESSIVITY PROPERTY is not satisfied.

2. (EFFICIENT MVM PROPERTY) This property is satisfied since one can multiply a Vandermonde matrix with an arbitrary vector in $O(n \log^2 n) = O(s')$ operations [11].

3. (EFFICIENT GRADIENT PROPERTY) By definition, each entry in a Vandermonde matrix is a polynomial (of degree at most $n-1$) in the entries of $\boldsymbol{\theta}$ and hence is also differentiable.

[15] More specifically, [19, Theorem 4.7] shows that one can achieve for any $\gamma > 0$, $r = \frac{n}{\exp(\gamma^6 (\log n)^{0.36})}$ and $\sigma \leq n^{7\gamma}$. In other words, their result only works for $r = n^{1-o(1)}$, which implies $s' \geq rn \geq n^{2-o(1)}$.

[16] Theorem 2.5 states $s = O(n \log n)$. As it turns out that result also implies a circuit with $s = O(n \log n)$ and $d = O(\log n)$. Now fix any $\varepsilon < 1/4$. Then if we pick $\sigma > n^{1/4-\varepsilon}$, then we have $s' \geq \sigma n \geq \Omega\left(n^{5/4-\varepsilon}\right)$. On the other hand, if we pick $\sigma \leq n^{1/4-\varepsilon}$, then Theorem 4.1 implies that $r \geq \Omega\left(\frac{n \log\log n}{\log n}\right)$, which means we in this case have $s' \geq 2rn \geq \Omega\left(n^2 \cdot \frac{\log\log n}{\log n}\right)$. Thus, over all choices of σ, we have $s' \geq \Omega\left(n^{5/4-\varepsilon}\right)$ as claimed.

4.5 Low-displacement rank matrices

We now consider a class of structured matrices that have been used in experiments in deep learning to address the practical questions that motivated Question 2.3.

We begin with the definition of a matrix having a displacement rank of r:

Definition 4.2. *A matrix* $\mathbf{W} \in \mathbb{F}^{n \times n}$ *has a* displacement rank *with respect to* $\mathbf{L}, \mathbf{R} \in \mathbb{F}^{n \times n}$*, if the* residual

$$\mathbf{E} = \mathbf{LW} - \mathbf{WR}$$

has rank r*.*

We would like to mention that for the above definition to be meaningful, the *displacement operators* $(\mathbf{L}, \mathbf{R})$ need to satisfy some non-trivial requirements for $\mathbf{W}$. E.g. if $\mathbf{L} = \mathbf{R} = \mathbf{I}$, then all matrices have displacement rank 0 with respect to $(\mathbf{I}, \mathbf{I})$. However, if we insists that $\mathbf{L}$ and $\mathbf{R}$ do not share any common eigenvalues, then in the above definition, every $\mathbf{E}$ corresponds to a unique matrix $\mathbf{W}$. For the rest of the section, we will make this assumption.

4.5.1 Some examples and arithmetic circuit complexity

Consider the following matrix (called a Cauchy matrix):

Definition 4.3 (Cauchy Matrix)**.** *Arbitrarily fix* $\boldsymbol{s}, \boldsymbol{t} \in \mathbb{F}^n$ *such that for every* $0 \le i, j < n$*,* $\boldsymbol{s}[i] \neq \boldsymbol{t}[j]$*,* $\boldsymbol{s}[i] \neq \boldsymbol{s}[j]$ *and* $\boldsymbol{t}[i] \neq \boldsymbol{t}[j]$ *and*

$$\mathbf{C}_n[i, j] = \frac{1}{\boldsymbol{s}[i] - \boldsymbol{t}[j]}.$$

It can be shown that this matrix has full rank. We next argue that the Cauchy matrix (Definition 4.3) has displacement rank 1 with respect to $\mathbf{L} = \mathrm{diag}(\boldsymbol{s})$ and $\mathbf{R} = \mathrm{diag}(\boldsymbol{t})$, where recall $\mathrm{diag}(\boldsymbol{x})$ is the diagonal matrix with $\boldsymbol{x}$ on its diagonal. Indeed, note that in this case we have $\mathrm{diag}(\boldsymbol{s})\mathbf{C}_n - \mathbf{C}_n\mathrm{diag}(\boldsymbol{t})$ is the all ones matrix.

Further, it turns out that the Vandermonde matrix (Definition 2.6) $\mathbf{V}_n^{(\boldsymbol{a})}$ for any $\boldsymbol{a} \in \mathbb{F}^n$ has displacement rank 1 with respect to $\mathbf{L} = \mathrm{diag}(\boldsymbol{a})$ and $\mathbf{R}$ being the *shift* matrix as defined next:

Definition 4.4 (Shift Matrix)**.** *The* shift *matrix* $\mathbf{Z} \in \mathbb{F}^{n \times n}$ *is defined by*

$$\mathbf{Z}[i, j] = \begin{cases} 1 & \text{if } i = j - 1 \\ 0 & \text{otherwise.} \end{cases}$$

(The reason the matrix $\mathbf{Z}$ is called the shift matrix is because when applied to the left or right of a matrix it shifts the row (or columns respectively) of the matrix.)

It is known how to compute $\mathbf{W}\boldsymbol{x}$ with arithmetic circuit complexity $\widetilde{O}(rn)$, where $\mathbf{W}$ has displacement rank at most r with respect to $\mathbf{L}, \mathbf{R}$ where these 'operators' are either shift or diagonal matrices. In fact De Sa et al. [18] show that as long as $\mathbf{L}$ and $\mathbf{R}$ are $O(1)$-quasiseparable (i.e. all sub-matrices strictly above or strictly below the main diagonal are $O(1)$-rank) then any matrix $\mathbf{W}$ that has rank r with respect to $(\mathbf{L}, \mathbf{R})$ has arithmetic circuit complexity of $\widetilde{O}(rn)$.

4.5.2 Low displacement rank matrices in deep learning literature

Low displacement rank (or LDR) matrices have actually been implemented in deep learning systems with some success in reducing the memory footprint and the time efficiency of inference [36, 38]. Here we give a very quick (and necessarily incomplete) overview of the main results of paper of Zhao et al. [43].

Zhao et al. consider LDR with respect to *any fixed* displacement operators $(\mathbf{L}, \mathbf{R})$ as long as

- Both $\mathbf{L}$ and $\mathbf{R}$ are non-singular diagonalizable matrices,
- $\mathbf{L}^q = a \cdot \mathbf{I}$ for some $1 \leq q \leq n$ and non-zero $a \in \mathbb{R}$,
- $\left(\mathbf{I} - a\mathbf{B}^q\right)$ is non-singular, and
- The eigenvalues of $\mathbf{R}$ are distinct in absolute values.

Zhao et al. fix the displacement operators $(\mathbf{L}, \mathbf{R})$ as above and consider one layer neural networks (as in our case) where the weight matrix $\mathbf{W}$ has $O(1)$-displacement rank with respect to $(\mathbf{L}, \mathbf{R})$.[17] Note that such matrices can be represented by just storing the residual $\mathbf{LW} - \mathbf{WR}$ and hence only needs $O(n)$ parameters overall. For the rest of this subsection, we will refer to these as LDR neural networks.

They show that for three well-studied properties of single layer neural networks, the one with LDR weight matrices as just as 'good' as arbitrary weight matrices. Arguing these results formally is out of scope for this survey so here we just given a very high level informal statements (and refer the reader to the paper [43] for the formal statements and their proofs):

1. The *universal approximation theorem* states that an LDR neural network can approximate any continuous function to within arbitrary precision over any point (i.e. under ℓ_∞ error norm).
2. The paper also shows that for *any* probability distribution over an n-dimensional ball, an LDR neural network can approximate any function (w.r.t. the probability distribution) with squared error $O(1/n^2)$. A similar result was shown for neural networks with arbitrary weight matrix, which we have already seen needs $\Omega(n^2)$ parameters (while the LDR neural network only needs $O(n)$ parameters as observed above).
3. Zhao et al. also show that one can compute the required gradients for the gradient descent algorithm (where roughly speaking the complexity of computing the gradients depends on the arithmetic circuit complexity of $\mathbf{L}$ and $\mathbf{R}$).[18]

One practical drawback in this setup is that one fixes $\mathbf{L}$ and $\mathbf{R}$ upfront. Thomas et al. have run experiments where one tries to learn the displacement operator $\mathbf{L}$ and $\mathbf{R}$ *along* with the residual matrix [38].

4.5.3 Coming back to Question 2.3

Unfortunately, low displacement rank matrices are not enough to answer Question 2.3 in the affirmative either.

[17] This means that during the learning phase, we already know $\mathbf{L}$ and $\mathbf{R}$ and we only need to learn the residual.

[18] At a high level this should not be surprising given the results in Section 3, though Zhao et al. do not utilize the generic connection we established in Section 3.

- (EXPRESSIVITY PROPERTY) It turns out that the full power of low displacement rank matrices w.r.t. $O(1)$-quasiseparable displacement matrices is not known– in other words, it is not known if these matrices satisfy EXPRESSIVITY PROPERTY. We conjecture that they do *not*. In a somewhat weak support of this conjecture, we note that the traditional low displacement operators are either a (non-zero) diagonal matrix $\mathbf{D}$ or (simple variants) of the shift matrix $\mathbf{Z}$ (the initial experimental results on LDR neural networks are for these displacement operators [36])– and in this case there are even diagonal matrices that have displacement rank $\Omega(n)$ with respect to these displacement operators (which means we have $s' \geq \Omega(n^2)$).

  Indeed, if at least one of $\mathbf{L}$ or $\mathbf{R}$ is a diagonal matrix, then we note that $\mathbf{L}-\mathbf{R}$ has all non-zero diagonal entries[19] and is lower/upper triangular and hence $\mathbf{W}=\mathbf{I}$ has displacement rank of $\Omega(n)$ with respect to such matrices. If both $\mathbf{L}$ and $\mathbf{R}$ are shift matrices, then we note that for a diagonal matrix $\mathbf{D}$, we have $\mathbf{E}=\mathbf{ZD}-\mathbf{DZ}=\mathbf{D}'\cdot\mathbf{Z}$, where elements of $\mathbf{D}'[i,i]=\mathbf{D}[i,i]-\mathbf{D}[i+1,i+1]$. Thus, if we choose $\mathbf{D}$ such that all the consecutive elements on the diagonal are different, then we have that rank of $\mathbf{E}$ is the same as rank of $\mathbf{Z}$ and thus, $\mathbf{D}$ will have displacement rank $\Omega(n)$ with respect to shift matrices.

- (EFFICIENT MVM PROPERTY) Results from [18] show that this property is satisfied when $\mathbf{L}$ and $\mathbf{R}$ are $O(1)$-quasiseparable matrices.

- (EFFICIENT GRADIENT PROPERTY) As mentioned earlier, [43] shows that this property is satisfied *if* $\mathbf{L}$ and $\mathbf{R}$ are fixed. Results in Section 3 imply EFFICIENT GRADIENT PROPERTY are satisfied as long as $\mathbf{W}$ has efficient matrix-vector multiplication.

4.6 Product of sparse matrices (in listing representation)

All of the classes of structured matrices that we have considered so far have all not been able to satisfy EXPRESSIVITY PROPERTY in Question 2.3. Next we consider the class of *product* of sparse matrices. De Sa et al. [18], showed that these can accurately capture $\mathbf{W}$ with small arithmetic circuits:

Theorem 4.5. *Let $\mathbf{W}$ be an $n\times n$ matrix such that matrix-vector multiplication of $\mathbf{W}$ times an arbitrary vector $\boldsymbol{v}$ can be represented as a linear arithmetic circuit C comprised of s gates (including inputs) and having depth d. Then we can represent $\mathbf{W}$ as a product of $d+1$ matrices each of which is $O(s)$ sparse.*

In fact [18] also proves a 'converse' of the above result (which means product of sparse matrices *exactly* capture the power of (linear) arithmetic circuits for linear maps). Before we present the proof of the above result, we remark that Theorem 4.5 and its converse in [18] are probably known but we have not been able to to find a reference that pre-dates [18]– if you are aware of a reference for the above theorem, please let the author know.

Proof of Theorem 4.5. We will represent C as a product of d matrices, each of size $s'\times s'$, where s' is the smallest power of 2 that is greater than or equal to s.

Define $w_1,\ldots w_d$ such that w_k represents the number of gates in the k'th layer of C (note that $s=n+\sum_{k=1}^{d} w_k$). Also, define $z_1,\ldots z_d$ such that $z_1=n$ and $z_k=w_{k-1}+z_{k-1}$ (z_k is the number of gates that have already been used by the time we get to layer k).

[19] If WLOG $\mathbf{L}=\mathbf{Z}$ and $\mathbf{R}=\mathbf{D}$, then the diagonal of $\mathbf{L}-\mathbf{R}$ is the diagonal of $\mathbf{D}$ and hence all non-zero by our assumption. If both $\mathbf{L}$ and $\mathbf{R}$ are diagonal matrices, i.e. $\mathbf{L}=\mathbf{D}_1$ and $\mathbf{R}=\mathbf{D}_2$, then $\mathbf{L}-\mathbf{R}=\mathbf{D}_1-\mathbf{D}_2$ and all these entries are non-zero since we assumed $\mathbf{L}$ and $\mathbf{R}$ do not share any eigenvalues.

Let g_i denote the i'th gate (and its output) of C ($0 \le i < s$), defined such that (where we want to multiply $\boldsymbol{v} = (v_0, \ldots, v_{n-1})$ with $\mathbf{W}$):

$$g_i = \begin{cases} v_i & 0 \le i < n \\ \alpha_i g_{i_1} + \beta_i g_{i_2} & n \le i < s \end{cases}$$

where i_1, i_2 are indices of gates in earlier layers.

For the k'th layer of C, we define the $s' \times s'$ matrix $\mathbf{W}_k$ such that it performs the computations of the gates in that layer. Define the i'th row of $\mathbf{W}_k$ to be:

$$\mathbf{W}_k[i:] = \begin{cases} \boldsymbol{e}_i^T & 0 \le i < z_k \\ \alpha_i \boldsymbol{e}_{i_1}^T + \beta_i \boldsymbol{e}_{i_2}^T & z_k \le i < z_k + w_k \\ 0 & i \ge z_k + w_k \end{cases}$$

For any $0 \le k \le d$, let $\boldsymbol{v_k}$ be vector

$$\boldsymbol{v}_k = \mathbf{W}_k \ldots \mathbf{W}_2 \mathbf{W}_1 \begin{bmatrix} \boldsymbol{v} \\ \mathbf{0} \end{bmatrix}.$$

We'd like to argue that $\boldsymbol{v}_d$ contains the outputs of all gates in C (i.e, the n values that make up $\mathbf{W}\boldsymbol{v}$). To do this we argue, by induction on k, that $\boldsymbol{v}_k$ is the vector whose first z_{k+1} entries are $g_0, g_1, \ldots, g_{(z_{k+1}-1)}$, and whose remaining entries are 0. The base case, $k = 0$ is trivial. Assuming this holds for the case $k-1$, and consider multiplying $\boldsymbol{v}_{k-1}$ by $\mathbf{W}_k$. The first z_k rows of $\mathbf{W}_k$ duplicate the first z_k entries of $\boldsymbol{v}_{k-1}$. The next w_k rows perform the computation of gates $g_{z_k}, \ldots, g_{(z_{k+1}-1)}$. Finally, the remaining rows pad the output vector with zeros. Therefore, $\boldsymbol{v}_k$ is exactly as desired.

The final matrix product will contain all n elements of the output, as desired. By left multiplying by some permutation matrix $\mathbf{P}$, we can reorder this vector such that the first n entries are exactly $\mathbf{W}\boldsymbol{v}$ (or more precisely we left multiply by a 'truncated' permutation matrix so that the final answer is exactly $\mathbf{W}\boldsymbol{v}$). One can now check that we have product of $d+1$ matrices each of which is $O(s)$ sparse, as desired. □

We are now ready to evaluate whether product of sparse matrices can answer Question 2.3 (spoiler alert: no!):

- (EXPRESSIVITY PROPERTY) If we assume that we only consider $\mathbf{W}$ that have circuit with depth $\widetilde{O}(1)$ (which capture most of the known efficient matrix-vector multiplication algorithms), then Theorem 4.5 shows that $s' = O(ds)$, which by our assumption on d is $\widetilde{O}(s)$, which means we have satisfied EXPRESSIVITY PROPERTY.
- (EFFICIENT MVM PROPERTY) If one uses the obvious algorithm (i.e. multiply successively by each of the $d+1$ matrices, each of which is $O(s)$-sparse), then one can compute the overall matrix vector multiplication in $O(ds) = O(s')$ operations. Thus, we also satisfy EFFICIENT MVM PROPERTY.
- (EFFICIENT GRADIENT PROPERTY) This property is not satisfied if we assume the listing representation for each of the sparse matrices (due to the same reason that a single sparse matrix in listing representation does not satisfy EFFICIENT GRADIENT PROPERTY).

We came close to answering Question 2.3 with product of sparse matrices– the only catch was that the listing representation of sparse matrices does not allow us to satisfy EFFICIENT GRADIENT PROPERTY. Next, we answer Question 2.3 in the positive by coming up with an alternative representation of sparse matrices that is differentiable.

5 Butterfly matrices

In this section, we will present a positive answer to Question 2.3. We start with taking a circuit/matrix-product view of the FFT in Section 5.1, which in turn motivates the definition of butterfly matrices in Section 5.2. Finally, we use Butterfly matrices to define the final class of matrices in Section 5.3, which we will show answer Question 2.3 in the affirmative.

5.1 Fast Fourier Transform (FFT)

As mentioned earlier, a vast majority of efficient matrix vector multiplication algorithms are equivalent to small (both in size and depth) linear arithmetic circuit. For example the FFT can be thought of as an efficient arithmetic circuit to compute the Discrete Fourier Transform (indeed when one converts the linear arithmetic circuit for FFT into a matrix decomposition, then each matrix in the decomposition is so called *Butterfly matrix*, with each block matrix in each factor being the same). For an illustration of this consider the DFT with $n = 4$ as illustrated in Figure 1.

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$$

Figure 1: DFT of order 4.

Figure 2 represent the arithmetic circuit corresponding to FFT with $n = 4$.

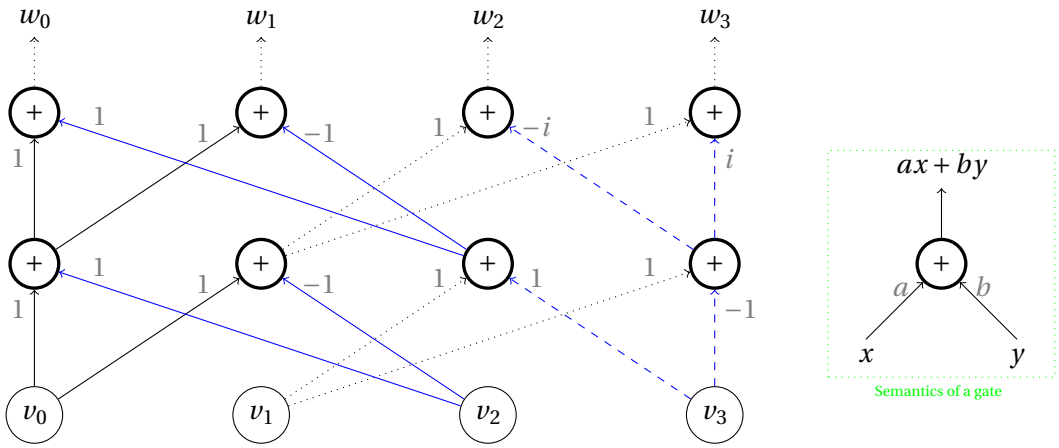

Figure 2: Arithmetic circuit for 4-DFT from Figure 1.

Finally, Figure 3 is representation of the arithmetic circuit of Figure 2 as a product of a butterfly matrix and (the bit-reversal) permutation.

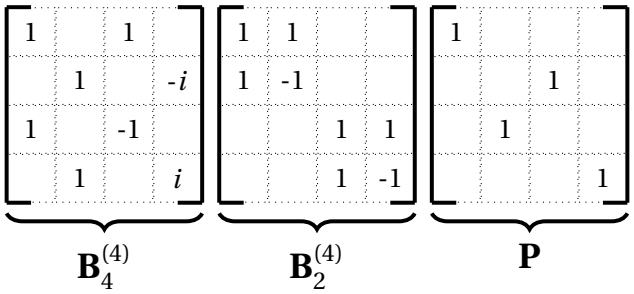

Figure 3: Decomposition of DFT of Figure 1 via the arithmetic circuit of Figure 2.

5.2 Butterfly matrices

Butterfly matrices, encoding the recursive divide-and-conquer structure of the fast Fourier transform (FFT) algorithm as illustrated in Figure 3, have long been used in numerical linear algebra [33, 29] and machine learning [30, 24, 31, 16, 13]. Here we define butterfly matrices, which we use as a building block for our hierarchy of kaleidoscope matrices.

Definition 5.1. *A **butterfly factor** of size $k \geq 2$ (denoted as $\mathbf{B}_k$) is a matrix of the form $\mathbf{B}_k = \begin{bmatrix} \mathbf{D}_1 & \mathbf{D}_2 \\ \mathbf{D}_3 & \mathbf{D}_4 \end{bmatrix}$ where each $\mathbf{D}_i$ is a $\frac{k}{2} \times \frac{k}{2}$ diagonal matrix. We restrict k to be a power of 2.*

Definition 5.2. *A **butterfly factor matrix** of size n with block size k (denoted as $\mathbf{B}_k^{(n)}$) is a block diagonal matrix of $\frac{n}{k}$ (possibly different) butterfly factors of size k:*

$$\mathbf{B}_k^{(n)} = \operatorname{diag}\left([\mathbf{B}_k]_1, [\mathbf{B}_k]_2, \ldots, [\mathbf{B}_k]_{\frac{n}{k}}\right)$$

Definition 5.3. *A **butterfly matrix** of size n (denoted as $\mathbf{B}^{(n)}$) is a matrix that can be expressed as a product of butterfly factor matrices: $\mathbf{B}^{(n)} = \mathbf{B}_n^{(n)} \mathbf{B}_{\frac{n}{2}}^{(n)} \ldots \mathbf{B}_2^{(n)}$. Equivalently, we may define $\mathbf{B}^{(n)}$ recursively as a matrix that can be expressed in the following form:*

$$\mathbf{B}^{(n)} = \mathbf{B}_n^{(n)} \begin{bmatrix} [\mathbf{B}^{(\frac{n}{2})}]_1 & 0 \\ 0 & [\mathbf{B}^{(\frac{n}{2})}]_2 \end{bmatrix}$$

(Note that $[\mathbf{B}^{(\frac{n}{2})}]_1$ and $[\mathbf{B}^{(\frac{n}{2})}]_2$ may be different.)

5.3 The kaleidoscope hierarchy

Using the building block of butterfly matrices, we formally define the kaleidoscope ($\mathcal{BB}^*$) hierarchy and prove its expressiveness. This class of matrices serves as a fully differentiable alternative to products of sparse matrices (Section 4.6), with similar expressivity. This family of matrices was defined by Dao et al. [17].

The building block for this hierarchy is the product of a butterfly matrix and the (conjugate) transpose of another butterfly matrix (which is simply a product of butterfly factors taken in the opposite order). Figure 4 visualizes the sparsity patterns of the butterfly factors in $\mathcal{BB}^*$, where the red and blue dots represent the allowed locations of nonzero entries.

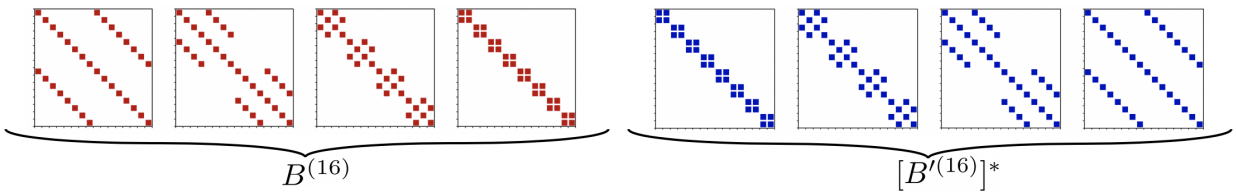

Figure 4: Visualization of the fixed sparsity pattern of the building blocks in $\mathcal{BB}^*$, in the case $n = 16$. The red and blue dots represent all the possible locations of the nonzero entries.

We would like to note that the sparsity pattern in a matrix in $\mathcal{BB}^*$ matches *exactly* the Beneš network [9, 8], which is a multistage circuit switching network. The goal in a switching network in Beneš network is to route n input connection to n output connection through a sequence of switches where the basic building block is a *cross-bar switch* (where each such switch can 'swap' two connections). It is known that the Beneš network can route *any permutation* from the input connection to the output connection by appropriately making the switch swap (or not) its two input connections. In our setup of $\mathcal{BB}^*$, we allow each 'switch' in a Beneš network to be replaced by an arbitrary 2×2 sub-matrix.

Definition 5.4 (Kaleidoscope hierarchy, kaleidoscope matrices)**.**

- *Define $\mathcal{B}$ as the set of all matrices that can be expressed in the form $\mathbf{B}^{(n)}$ (for some n).*
- *Define $\mathcal{BB}^*$ as the set of matrices $\mathbf{M}$ of the form $\mathbf{M} = \mathbf{M}_1\mathbf{M}_2^*$ for some $\mathbf{M}_1, \mathbf{M}_2 \in \mathcal{B}$.*
- *Define $(\mathcal{BB}^*)^w$ as the set of matrices $\mathbf{M}$ that can be expressed as $\mathbf{M} = \mathbf{M}_w \dots \mathbf{M}_2\mathbf{M}_1$, with each $\mathbf{M}_i \in \mathcal{BB}^*$ $(1 \le i \le w)$. (The notation w represents **width**.)*
- *Define $(\mathcal{BB}^*)_e^w$ as the set of $n \times n$ matrices $\mathbf{M}$ that can be expressed as $\mathbf{M} = \mathbf{SES}^T$ for some $en \times en$ matrix $\mathbf{E} \in (\mathcal{BB}^*)^w$, where $\mathbf{S} \in \mathbb{F}^{n \times en} = \begin{bmatrix} \mathbf{I}_n & 0 & \dots & 0 \end{bmatrix}$ (i.e. $\mathbf{M}$ is the upper-left corner of $\mathbf{E}$). (The notation e represents **expansion** relative to n.)*
- *$\mathbf{M}$ is a **kaleidoscope matrix**, abbreviated as **K-matrix**, if $M \in (\mathcal{BB}^*)_e^w$ for some w and e.*

The *kaleidoscope hierarchy*, or $(\mathcal{BB}^*)$ hierarchy, refers to the families of matrices $(\mathcal{BB}^*)_e^1 \subseteq (\mathcal{BB}^*)_e^2 \subseteq \dots$, for a fixed expansion factor e. Each butterfly matrix can represent the identity matrix, so $(\mathcal{BB}^*)_e^w \subseteq (\mathcal{BB}^*)_e^{w+1}$. Dao et al. [17] show that the inclusion is proper.

Efficiency in space and speed. Each matrix in $(\mathcal{BB}^*)_e^w$ is a product of $2w$ total butterfly matrices and transposes of butterfly matrices, each of which is in turn a product of $\log(ne)$ factors with $2ne$ nonzeros (NNZ) each. Therefore, each matrix in $(\mathcal{BB}^*)_e^w$ has $4wne\log(ne)$ parameters and a matrix-vector multiplication algorithm of complexity $O(wne\log ne)$ (by multiplying the vector with each sparse factor sequentially).

Difference from family of matrices in Section 4.6. We note that the K-matrices are similar to the family of matrices considered in Section 4.6 in that they are also product of sparse matrices. The main difference is that each matrix in the product in addition to being sparse is also *structured*– i.e. we know upfront where all the non-zero elements in each factor in a K-matrix will be. This allows us to create a differentiable representation for sparse matrices, which was the missing part of the family of product of (general) sparse matrices.

5.3.1 Answering Question 2.3

We state the main theoretical result, namely, the ability to capture general transformations, expressed as low-depth linear arithmetic circuits, in the $\mathscr{B}\mathscr{B}^*$ hierarchy. This result is recorded in Theorem 5.5.

Theorem 5.5. *Let* $\mathbf{M}$ *be an* $n \times n$ *matrix such that matrix-vector multiplication of* $\mathbf{M}$ *times an arbitrary vector* $\boldsymbol{v}$ *can be represented as a linear arithmetic circuit* C *comprised of* s *gates (including inputs) and having depth* d*. Then,* $\mathbf{M} \in (\mathscr{B}\mathscr{B}^*)^{O(d)}_{O(\frac{s}{n})}$.

Before we prove Theorem 5.5, we note that it is sufficient to show that K-matrices answer Question 2.3 in the affirmative:

1. (EXPRESSIVITY PROPERTY) Theorem 5.5 along with the observation on number of parameters needed to represent a matrix in $(\mathscr{B}\mathscr{B}^*)^w_e$ implies that we have $s' = O\left(d \cdot \frac{s}{n} \cdot n \log\left(\frac{s}{n} \cdot n\right)\right) = O(ds \log s)$. Thus, under the assumption of $d = \widetilde{O}(1)$, we have that $s' = \widetilde{O}(s)$, as desired.

2. (EFFICIENT MVM PROPERTY) Again by Theorem 5.5 along observation on number of operations needed to do matrix-vector multiplication for a matrix in $(\mathscr{B}\mathscr{B}^*)^w_e$ (and using the calculations from the previous bullet), we get that the matrix-vector multiplication takes $O(s')$ operations, as desired.

3. (EFFICIENT GRADIENT PROPERTY) Finally, since we know the locations of the non-zero elements (which form the parameters for K-matrices), it is not too hard to see that each entry in $\mathbf{W}_{\boldsymbol{\theta}}$ is a polynomial in the entries of $\boldsymbol{\theta}$. Since a polynomial in $\boldsymbol{\theta}$ is differentiable, this means EFFICIENT GRADIENT PROPERTY is satisfied as well.

Proof of Theorem 5.5. To prove Theorem 5.5, we make use of the following two theorems.

Theorem 5.6. *Let* $\mathbf{P}$ *be an* $n \times n$ *permutation matrix (with* n *a power of 2). Then* $\mathbf{P} \in \mathscr{B}\mathscr{B}^*$.

Theorem 5.7. *Let* $\mathbf{S}$ *be an* $n \times n$ *matrix of* s *NNZ. Then* $\mathbf{S} \in (\mathscr{B}\mathscr{B}^*)^{4\lceil \frac{s}{n} \rceil}_4$.

We first give an overview of how the above two results imply Theorem 5.5 and then briefly outline how one can prove the two results above. First, we note that (proof of) Theorem 4.5 implies that given any $\mathbf{W}$ with arithmetic circuit of size s and depth d, we can represent $\mathbf{W}$ as a product of d many $O(s)$-sparse matrices and a permutation matrix. Thus, we can decompose $\mathbf{W}$ as product of $O(d)$ K-matrices. Then Theorem 5.5 follows from the simple observation that membership in the family of K-matrices is closed under multiplication.

Theorem 5.6 essentially follows from the known fact that a Beneš network can route an arbitrary permutation (see [17] for a self-contained proof in the language of K-matrices).

Theorem 5.7 follows by showing that any n-sparse matrix is in $(\mathscr{B}\mathscr{B}^*)^4$ and the fact that membership in the family of K-matrices is closed under addition. To show the inclusion of an n-sparse matrix, Dao et al. [17] show that any n-sparse matrix $\mathbf{S}$ can be decomposed as $\mathbf{P}_1\mathbf{H}\mathbf{P}_2\mathbf{V}\mathbf{P}_3$, where $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$ are permutation matrices (which by Theorem 5.6 are in $\mathscr{B}\mathscr{B}^*$). $\mathbf{H}$ is *horizontal step matrix*, which obeys a 'Lipschitz-like' condition. Each column of a horizontal step matrix can have at most one non-zero entry, and given two non-zero columns k apart, the non-zero entry in the right column must be between 0 and k rows below the non-zero entry in the left column. Note that to show that a matrix is a horizontal step matrix, it is sufficient to argue that this condition holds for each pair of neighboring non-zero columns.

The matrix $\mathbf{V}$ is such that its transpose is a horizontal step matrix. Dao et al. [17] show that any horizontal step matrix is in $\mathscr{B}$. Combining all of these, we have that $\mathbf{S} \in (\mathscr{B}\mathscr{B}^*)(\mathscr{B})(\mathscr{B}\mathscr{B}^*)(\mathscr{B}^*)(\mathscr{B}\mathscr{B}^*) \subseteq (\mathscr{B}\mathscr{B}^*)^5$. Dao et al. [17] observe that with bit more careful analysis we can show inclusion in $(\mathscr{B}\mathscr{B}^*)^4$. We refer the interested reader to [17] for the proof details.

6 Open Questions

We conclude by present two open questions (the first one being a specific technical question and the other one being a bit more vague):

1. There is one unsatisfactory aspect to the results in Section 5.3, i.e. the number of parameters needed to specify the family of K-matrices that capture matrices with arithmetic circuit of size s and depth d is $O(sd\log s)$. In particular, the dependence on d is not ideal, which leads to the following:

   > **Open Question 6.1.** *Is it possible to answer Question 2.3 in the affirmative with a family that uses $s' = \widetilde{O}(s)$ many parameters to capture all matrices with arithmetic circuits of size s (irrespective of the depth d)?*

2. As mentioned earlier, low rank approximation is ubiquitous in machine learning (and numerical linear algebra more generally). One intriguing possibility is whether K-matrices can replace low rank matrices in these applications? Currently, the main technical stumbling block is solving the following:

   > **Open Question 6.2.** *Does there exist an efficient algorithm that solves the following problem– given an arbitrary matrix $\mathbf{M} \in \mathbb{F}^{n\times n}$ and parameters w and e, find the matrix $\mathbf{W} \in (\mathscr{B}\mathscr{B}^*)_e^w$ that is closest (or 'close enough') to $\mathbf{M}$ (say in Frobenius norm)?*

   We note that for low rank matrices, the SVD solves the above question. Thus, the question is asking whether we can be design the 'SVD for K-matrices'? Partial progress on a variant of the above question was made recently in [15].

Acknowledgments

The material in Sections 2 and 3 are based on notes for AR's Open lectures for PhD students in computer science at University of Warsaw titled *(Dense Structured) Matrix Vector Multiplication* in May 2018– we would like to thank University of Warsaw's hospitality. The material in Section 5 is based on Dao et al. [17].

We would like to thank Tri Dao, Albert Gu and Chris Ré for many illuminating discussions during our collaborations around these topics.

We would like to thank an anonymous reviewer whose comments improved the presentation of the survey and we thank Jessica Grogan for a careful read of an earlier draft of this survey.

AR is supported in part by NSF grant CCF-1763481.

References

[1] Josh Alman. Kronecker products, low-depth circuits, and matrix rigidity. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 772–785. ACM, 2021.

[2] Josh Alman and Lijie Chen. Efficient construction of rigid matrices using an NP oracle. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 1034–1055. IEEE Computer Society, 2019.

[3] Josh Alman and R. Ryan Williams. Probabilistic rank and matrix rigidity. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 641–652. ACM, 2017.

[4] M. Anthony and P.L. Bartlett. *Neural Network Learning: Theoretical Foundations*. Neural Network Learning: Theoretical Foundations. Cambridge University Press, 2009.

[5] Solon Barocas, Moritz Hardt, and Arvind Narayanan. *Fairness and Machine Learning*. fairmlbook.org, 2019. http://www.fairmlbook.org.

[6] Walter Baur and Volker Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22(3):317–330, 1983.

[7] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In Madeleine Clare Elish, William Isaac, and Richard S. Zemel, editors, *FAccT '21: 2021 ACM Conference on Fairness, Accountability, and Transparency, Virtual Event / Toronto, Canada, March 3-10, 2021*, pages 610–623. ACM, 2021.

[8] V.E. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic*. ISSN. Elsevier Science, 1965.

[9] V. E. Beneš. Optimal rearrangeable multistage connecting networks. *The Bell System Technical Journal*, 43(4):1641–1656, 1964.

[10] Amey Bhangale, Prahladh Harsha, Orr Paradise, and Avishay Tal. Rigid matrices from rectangular pcps or: Hard claims have complex proofs. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 858–869. IEEE, 2020.

[11] Peter Bürgisser, Michael Clausen, and Mohammad A. Shokrollahi. *Algebraic complexity theory*, volume 315. Springer Science & Business Media, 2013.

[12] Emmanuel J. Candès, Xiaodong Li, Yi Ma, and John Wright. Robust principal component analysis? *J. ACM*, 58(3), June 2011.

[13] Krzysztof Choromanski, Mark Rowland, Wenyu Chen, and Adrian Weller. Unifying orthogonal Monte Carlo methods. In *International Conference on Machine Learning*, pages 1203–1212, 2019.

[14] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[15] Tri Dao, Beidi Chen, Nimit Sharad Sohoni, Arjun D. Desai, Michael Poli, Jessica Grogan, Alexander Liu, Aniruddh Rao, Atri Rudra, and Christopher Ré. Monarch: Expressive structured matrices for efficient and accurate training. *CoRR*, abs/2204.00595, 2022.

[16] Tri Dao, Albert Gu, Matthew Eichhorn, Atri Rudra, and Christopher Ré. Learning fast algorithms for linear transforms using butterfly factorizations. In *The International Conference on Machine Learning (ICML)*, 2019.

[17] Tri Dao, Nimit Sharad Sohoni, Albert Gu, Matthew Eichhorn, Amit Blonder, Megan Leszczynski, Atri Rudra, and Christopher Ré. Kaleidoscope: An efficient, learnable representation for all structured linear maps. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[18] Christopher De Sa, Albert Gu, Rohan Puttagunta, Christopher Ré, and Atri Rudra. A two-pronged progress in structured dense matrix vector multiplication. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1060–1079, 2018.

[19] Zeev Dvir and Allen Liu. Fourier and circulant matrices are not rigid. *Theory of Computing*, 16(20):1–48, 2020.

[20] Charles M. Fiduccia. *On the algebraic complexity of matrix multiplication*. PhD thesis, Brown University, 1973. URL: `http://cr.yp.to/bib/entries.html#1973/fiduccia-matrix`.

[21] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations (ICLR)*, 2019.

[22] Sasha Golovnev. A course on matrix rigidity, 2020. https://golovnev.org/rigidity/. Accessed August 15, 2021.

[23] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Li Wang, Gang Wang, Jianfei Cai, and Tsuhan Chen. Recent advances in convolutional neural networks. *Pattern Recognition*, 77:354–377, 2018.

[24] Li Jing, Yichen Shen, Tena Dubcek, John Peurifoy, Scott Skirlo, Yann LeCun, Max Tegmark, and Marin Soljačić. Tunable efficient unitary neural networks (eunn) and their application to rnns. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1733–1741. JMLR. org, 2017.

[25] Thomas Kailath, Sun-Yuan Kung, and Martin Morf. Displacement ranks of matrices and linear equations. *Journal of Mathematical Analysis and Applications*, 68(2):395–407, 1979.

[26] Thomas Kailath and Ali H. Sayed. Displacement structure: Theory and applications. *SIAM Review*, 37(3):297–386, 1995.

[27] E. Kaltofen. Computational differentiation and algebraic complexity theory. In *C. H. Bischof, A. Griewank, and P. M. Khademi, editors, Workshop Report on First Theory Institute on Computational Differentiation, volume ANL/MCS-TM-183 of Tech. Rep., Argonne, Illinois*, pages 28–30, New York, NY, USA, 1993. Association for Computing Machinery. http://kaltofen.math.ncsu.edu/bibliography/93/Ka93_diff.pdf.

[28] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436– 444, 2015.

[29] Yingzhou Li, Haizhao Yang, Eileen R. Martin, Kenneth L. Ho, and Lexing Ying. Butterfly factorization. *Multiscale Modeling & Simulation*, 13(2):714–732, 2015.

[30] Michael Mathieu and Yann LeCun. Fast approximation of rotations and Hessians matrices. *arXiv preprint arXiv:1404.7195*, 2014.

[31] Marina Munkhoeva, Yermek Kapushev, Evgeny Burnaev, and Ivan Oseledets. Quadrature-based features for kernel approximation. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 9165–9174. Curran Associates, Inc., 2018.

[32] Victor Y. Pan. *Structured Matrices and Polynomials: Unified Superfast Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[33] D. Stott Parker. Random butterfly transformations with applications in computational linear algebra. Technical report, UCLA, 1995.

[34] R. Paturi and P. Pudlák. Circuit lower bounds and linear codes. *Journal of Mathematical Sciences*, 134:2425– 2434, 2006.

[35] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green AI. *arXiv preprint arXiv:1907.10597*, 2019.

[36] Vikas Sindhwani, Tara N. Sainath, and Sanjiv Kumar. Structured transforms for small-footprint deep learning. In *Advances in Neural Information Processing Systems*, pages 3088–3096, 2015.

[37] G. Szegö. *Orthogonal Polynomials*. Number v. 23 in American Mathematical Society colloquium publications. American Mathematical Society, 1967.

[38] Anna T. Thomas, Albert Gu, Tri Dao, Atri Rudra, and Christopher Ré. Learning compressed transforms with low displacement rank. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 9066–9078, 2018.

[39] Joseph Tsidulko. Google showcases on-device artificial intelligence breakthroughs at I/O. *CRN*, 2019.

[40] Madeleine Udell and Alex Townsend. Why are big data matrices approximately low rank? *SIAM Journal on Mathematics of Data Science*, 1(1):144–160, 2019.

[41] Leslie G. Valiant. Graph-theoretic arguments in low-level complexity. In Jozef Gruska, editor, *Mathematical Foundations of Computer Science 1977*, pages 162–176, Berlin, Heidelberg, 1977. Springer Berlin Heidelberg.

[42] Xiyu Yu, Tongliang Liu, Xinchao Wang, and Dacheng Tao. On compressing deep models by low rank and sparse decomposition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[43] Liang Zhao, Siyu Liao, Yanzhi Wang, Zhe Li, Jian Tang, and Bo Yuan. Theoretical properties for neural networks with weight matrices of low displacement rank. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 4082–4090. PMLR, 06–11 Aug 2017.