LINEAR ALGEBRAIC METHODS IN NEURAL NETWORKS

[1]Ms.R.Divya
Assistant Professor, Department of Mathematics,
Sri Bharathi Engineering College for Women, Pudukkottai-622 303 , Tamilnadu, India.
Email: [1]rdivya2610@gmail.com

Abstract **- Neural networks have emerged as powerful tools for solving complex problems in various domains, ranging from image recognition to natural language processing. Understanding the mathematical foundations of neural networks is crucial for optimizing their performance and unlocking their full potential. This paper focuses on the application of linear algebraic methods in the analysis and enhancement of neural networks. A comprehensive review of matrix decompositions, such as Singular Value Decomposition (SVD) and Eigen value Decomposition, is presented in the context of neural networks. These techniques provide insights into the network's structure, aiding in model interpretation and identifying critical features that contribute to its performance. Additionally, the paper discusses how these methods can be employed for regularization, dimensionality reduction, and feature extraction in neural networks. Finally, practical applications of linear algebraic methods in neural networks are illustrated through case studies, demonstrating their efficacy in tasks such as transfer learning, adversarial robustness, and model compression. The paper concludes with a discussion on the potential avenues for future research in leveraging linear algebra to advance the field of neural network design and optimization.**

Keywords---Neural Networks, Linear Algebra,Matrix representation, Matrix Decomposition, Singular Value Decomposition(SVD), Computational efficiency.

I.INTRODUCTION

Neural networks have become a cornerstone of modern artificial intelligence, revolutionizing various fields including computer vision, natural language processing, and reinforcement learning. These networks, inspired by the structure and function of the human brain, consist of interconnected layers of neurons capable of learning complex patterns and relationships from data.

While neural networks exhibit remarkable performance in many tasks, understanding their inner workings and optimizing their performance remains a challenging endeavor. At the heart of neural network theory lies linear algebra, a branch of mathematics concerned with vector spaces and linear transformations. The application of linear algebraic methods in neural networks provides a rigorous framework for analyzing their behavior, interpreting their decisions, and enhancing their capabilities. By representing neural network operations in terms of matrices and vectors, we can leverage powerful mathematical tools to gain insights into their structure and dynamics.

This paper aims to explore the role of linear algebraic methods in advancing the theory and practice of neural networks. We begin by providing an overview of neural network architecture, highlighting the flow of information through layers of neurons and the mathematical operations involved in processing input data. Emphasis is placed on the non-linear transformations introduced by activation functions, which play a crucial role in enabling neural networks to model complex relationships.

Next, we delve into the matrix representations of neural network operations, demonstrating how concepts from linear algebra can be used to succinctly describe the computations performed by the network. We explore matrix decompositions such as Singular Value Decomposition (SVD) and Eigenvalue Decomposition, showcasing their utility in model interpretation, regularization, and dimensionality reduction.

The intersection of linear algebra and optimization is then explored in the context of neural network training. We discuss gradient descent variants and their connection to linear algebraic operations, highlighting the importance of efficient optimization techniques for training deep neural networks. Additionally, we investigate the role of weight initialization strategies and their impact on the convergence and generalization of neural network models.

Throughout the paper, we provide practical examples and case studies illustrating the application of linear algebraic methods in neural network design and optimization. Topics such as transfer learning, adversarial robustness, and model compression are discussed, demonstrating how linear algebra can be leveraged to address real-world challenges in machine learning.

This paper serves as a comprehensive exploration of the synergy between linear algebra and neural networks. By leveraging the rich mathematical framework provided by linear algebra, we can gain deeper insights into the behavior of neural networks and develop more efficient and robust learning algorithms. The integration of linear algebraic methods paves the way for further advancements in the field of neural network research and holds promise for unlocking the full potential of artificial intelligence.

II.SINGULAR VALUE DECOMPOSITION

The singular value decomposition (SVD) of a matrix is a decomposition of the matrix into a product of an orthogonal matrix, a diagonal matrix, and another orthogonal matrix. It is one of the most powerful ideas in linear algebra. However, to understand it fully one must first understand certain facts about symmetric matrices. Thus, our first section will show that all symmetric matrices are orthogonally diagonalizable. Not only can we construct a basis of eigenvectors for any symmetric matrix, but the the matrix formed out of these vectors, P, will be an orthogonal matrix! We will then make this relationship between orthogonal diagonalization and symmetric matrices even tighter; a matrix is orthogonally diagonalizable if and only if it is a symmetric matrix. This result is known as the spectral theorem.

Definition: A symmetric matrix is a n × n matrix A that is equal to its transpose. This means that for all 1 ≤ i, j ≤ n $a_{ij} = a_{ji}$.

Definition: A matrix A is orthogonally diagonalizable if there exists an orthogonal matrix P and a diagonal matrix D such that $A = P\,DP^{-1} = PDP^{T}$

Definition: For a symmetric n × n matrix A, we define a spectral decomposition of A as being a sum of the form

$$A = \lambda_1 u_1 u_1^T + \lambda_2 u_2 u_2^T + \cdots .. + \lambda_n u_n u_n^T$$

where P = $[u_1, u_2, ..., u_n]$ is an orthogonal set of unit eigenvectors, and $\lambda_1, \lambda_2, ..., \lambda_n$ are the eigenvalues of A corresponding to P. The spectral decomposition is in fact found by orthogonally diagonalizing A.

Theorem: Let A be any $m \times n$ matrix with rank r. Then, $A = U\Sigma V^T$ where U is an $m \times m$ orthogonal matrix, V is an $n \times n$ orthogonal matrix, and Σ is an m × n matrix such that

$$\Sigma = \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix}$$

where D is a r × r diagonal matrix. The remaining $m - r$ rows and $n - r$ columns of Σ will be 0. D will be the first r non-zero singular values of A, (σ_1, . . . , σ_r), such that

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0$$

We call $A = U\Sigma V^T$ a **singular value decomposition** of A.

The power of the singular value decomposition is that it exists for any matrix without restrictions. Because of this, the applications of the singular value decomposition are extremely powerful for data analysis.

III.NEURAL NETWORKS

A neural network is composed of neurons and edges with the neurons usually organized in layers and the directed edges connecting neurons from one layer to the next. We can think of neurons as variables with assigned values which we calculate through "forward propagation" which will be defined later. They are also called activation units. We can think of edges as variables whose value indicates how strongly one neuron influences another. The weights will serve as a type of scalar to the neuron it recieves.

These edges' values will be used to define functions that take the values of the neurons in one layer and use these to define values in the next layer. There will be a pre-determined number of layers in the network and the activation units in the final layer will signify something about the data inputted into the neural network. For example, suppose we have a data point with two variables and we want to classify the point as "on" or "off". Consider figure 1 which displays a neural network with one "hidden layer", the layers that do not contain input or output neurons and with three neurons inside this layer.

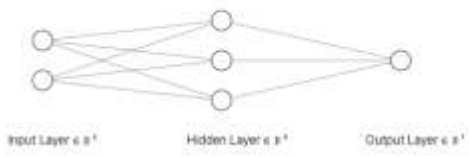
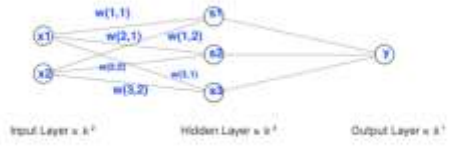


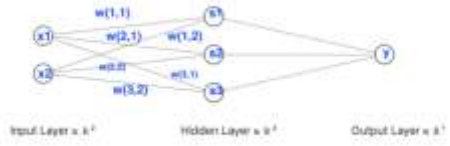
Figure 1. Structure of a Neural Network

The depth of the network is equal to the total number of layers in the network. Each layer will also have a width which is based on the number of neurons at each layer. We call the value of the edges that connect neurons to different layers, the weights of the network. The weights are used to define a function that uses one layer to define how one input neuron becomes another input neuron. We have x_1, x_2 as the input neurons, where w_{ij} represents the weights applied to the them. Also S_1, S_2, S_3 are the neurons at the hidden layer, and y is the output neuron. Consider figure



How do these weights transform the neurons? The best way to understand this is by viewing a neural network as simply layers of matrix-vector multiplication composed together.

If we have a data set the entire data set will usually be a data matrix X, where each vector is a data point. In our previous example, each data point would have two variables, x_1 and x_2. Thus, we can think of each layer of neurons as a vector. If a layer has 3 neurons, it would be represented as a vector with dimension 3. In a fully connected layer, there is an edge between each input neuron and each output neuron. In this case, we can represent the edges together as a matrix as well. We will call this the weight matrix. Thus, the action of the weights on the first layer becomes



$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 \\ w_{21}x_1 + w_{22}x_2 \\ w_{31}x_1 + w_{32}x_2 \end{bmatrix}$$

which would then undergo another matrix multiplication to produce the output neuron y.

We previously only described the interactions between neurons and edges as matrix-transformation. However, neural networks will be made up of non linear transformations. The goal of many neural networks is to identify complicated patterns to solve complicated problems. Having our functions limited to be linear functions would severely restrict the ability for neural networks to identify complicated patterns that will most likely not be linear. Therefore, at each layer we introduce, non linear activation functions which transform our linear functions into non linear functions. Consider the activation function σ as $\mathbb{R} \to \mathbb{R}$. Let $\sigma_b : \mathbb{R}^n \to \mathbb{R}$ where $b \in \mathbb{R}^n$. We now describe the interaction between neurons and edges as an non linear function becomes

$$\sigma_b\left(\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 \\ w_{21}x_1 + w_{22}x_2 \\ w_{31}x_1 + w_{32}x_2 \end{bmatrix}\right)$$

$$= \sigma_b\left(\begin{bmatrix} w_{11}x_1 + w_{12}x_2 \\ w_{21}x_1 + w_{22}x_2 \\ w_{31}x_1 + w_{32}x_2 \end{bmatrix}\right)$$

$$= \begin{bmatrix} \sigma(w_{11}x_1 + w_{12}x_2 - b_1) \\ \sigma(w_{21}x_1 + w_{22}x_2 - b_2) \\ \sigma(w_{31}x_1 + w_{32}x_2 - b_3) \end{bmatrix}$$

where σ_b maps the dimension of the output neurons to the same dimension. Thus if we have $y_1, \ldots y_n$ output neurons at each layer, we have

$$\sigma_b\left(\begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}\right) = \begin{bmatrix} \sigma(y_1 - b_1) \\ \vdots \\ \sigma(y_n - b_n) \end{bmatrix}$$

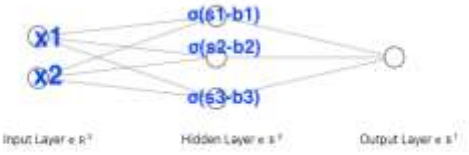


Figure 2. Non Linear Transformation

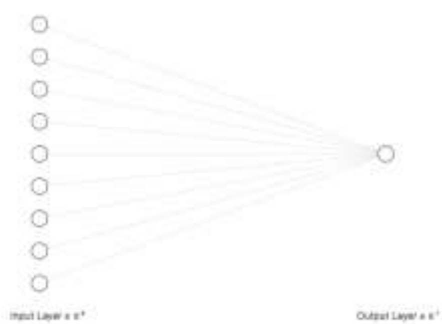
Example: Let's consider a helpful, but slightly unrealistic example. Suppose we have images representing two numbers: a 1 and a 0. A one would be a 3 × 3 matrix with values down the middle. A 0 will also be a 3 × 3 matrix but with values all the way across the perimeter. Let's consider that our data set has just a 1 and a 2.

$$1 = \begin{bmatrix} 0 & 5 & 0 \\ 0 & 8 & 0 \\ 0 & 1 & 0 \end{bmatrix}, 0 = \begin{bmatrix} 1 & 3 & 4 \\ 1 & 0 & 5 \\ 3 & 7 & 6 \end{bmatrix}$$

Note that if we vectorize both matrices, which is common in neural networks, the number in vector for will be

$$1 = \begin{bmatrix} 0 \\ 5 \\ 0 \\ 0 \\ 8 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, 0 = \begin{bmatrix} 1 \\ 3 \\ 4 \\ 1 \\ 0 \\ 5 \\ 3 \\ 7 \\ 6 \end{bmatrix}$$

Imagine we constructed a two-layer neural network, with 9 input neurons and one output neuron.



As explained later, the values of our weights will be randomly initialized. Nevertheless, their values would determine the value of the neurons in the final layer.

$$\sigma\left([4\ 4\ 3\ 3\ 1\ 6\ 7\ 9\ 10] \begin{bmatrix} 0 \\ 5 \\ 0 \\ 0 \\ 8 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \right) = \sigma([37])$$

$$\sigma\left([4\ 4\ 3\ 3\ 1\ 6\ 7\ 9\ 10] \begin{bmatrix} 1 \\ 3 \\ 4 \\ 1 \\ 0 \\ 5 \\ 3 \\ 7 \\ 6 \end{bmatrix} \right) = \sigma([205])$$

Both the non linear activation function and the bias vector work in tandem to improve our network's ability to make accurate predictions on complicated problems.

Applying the activation function first introduces this non linearity to our function, allowing our network to recognize more complex patterns. Secondly, together with the bias vector it helps normalize the values of neurons between a certain range. The bias vector helps determine the cut-off in how neurons will transitions between a certain range.

For example, consider a commonly used activation function where each neuron is scaled to a value between 0 and 1.

$$\sigma = \frac{1}{1 + e^{-\beta x}}$$

IV.CONVOLUTIONAL NEURAL NETWORKS

Convolution as a Sliding Dot Product:

Particularly for image-processing, most neural networks have multiple layers that involve convolution before they reach fully-connected layers. The purpose of convolutional layers is to extract features from the input image. The input layer will be some input image which can be represented with an m × n matrix A where each entry corresponds to a pixel in the image. This is the standard for grey-scale pictures. However, for color images where we are using the RGB color model, each RGB component of the image is represented by a matrix. Viewing these three separate m×n matrices as one object, we obtain a higher-dimensional version of a matrix called a tensor.

Consider the case of a grey-scale image represented by the following matrix

$$A = \begin{bmatrix} 3 & 3 & 2 & 1 & 0 & 5 \\ 0 & 0 & 1 & 3 & 1 & 6 \\ 3 & 1 & 2 & 2 & 3 & 7 \\ 2 & 0 & 0 & 2 & 2 & 8 \\ 2 & 0 & 0 & 0 & 1 & 9 \end{bmatrix}$$

We can consider the following 9 different sub-matrices $A_1, \ldots A_{12}$respectively

$$\begin{pmatrix} 3 & 3 & 2 & 1 & 0 & 5 \\ 0 & 0 & 1 & 3 & 1 & 6 \\ 3 & 1 & 2 & 2 & 3 & 7 \\ 2 & 0 & 0 & 2 & 2 & 8 \\ 2 & 0 & 0 & 0 & 1 & 9 \end{pmatrix} \begin{pmatrix} 3 & 3 & 2 & 1 & 0 & 5 \\ 0 & 0 & 1 & 3 & 1 & 6 \\ 3 & 1 & 2 & 2 & 3 & 7 \\ 2 & 0 & 0 & 2 & 2 & 8 \\ 2 & 0 & 0 & 0 & 1 & 9 \end{pmatrix}$$

$$\begin{pmatrix} 3 & 3 & 2 & 1 & 0 & 5 \\ 0 & 0 & 1 & 3 & 1 & 6 \\ 3 & 1 & 2 & 2 & 3 & 7 \\ 2 & 0 & 0 & 2 & 2 & 8 \\ 2 & 0 & 0 & 0 & 1 & 9 \end{pmatrix} \begin{pmatrix} 3 & 3 & 2 & 1 & 0 & 5 \\ 0 & 0 & 1 & 3 & 1 & 6 \\ 3 & 1 & 2 & 2 & 3 & 7 \\ 2 & 0 & 0 & 2 & 2 & 8 \\ 2 & 0 & 0 & 0 & 1 & 9 \end{pmatrix}$$

$$\begin{pmatrix} 3 & 3 & 2 & 1 & 0 & 5 \\ 0 & 0 & 1 & 3 & 1 & 6 \\ 3 & 1 & 2 & 2 & 3 & 7 \\ 2 & 0 & 0 & 2 & 2 & 8 \\ 2 & 0 & 0 & 0 & 1 & 9 \end{pmatrix} \begin{pmatrix} 3 & 3 & 2 & 1 & 0 & 5 \\ 0 & 0 & 1 & 3 & 1 & 6 \\ 3 & 1 & 2 & 2 & 3 & 7 \\ 2 & 0 & 0 & 2 & 2 & 8 \\ 2 & 0 & 0 & 0 & 1 & 9 \end{pmatrix}$$

$$\begin{pmatrix} 3 & 3 & 2 & 1 & 0 & 5 \\ 0 & 0 & 1 & 3 & 1 & 6 \\ 3 & 1 & 2 & 2 & 3 & 7 \\ 2 & 0 & 0 & 2 & 2 & 8 \\ 2 & 0 & 0 & 0 & 1 & 9 \end{pmatrix} \quad \begin{pmatrix} 3 & 3 & 2 & 1 & 0 & 5 \\ 0 & 0 & 1 & 3 & 1 & 6 \\ 3 & 1 & 2 & 2 & 3 & 7 \\ 2 & 0 & 0 & 2 & 2 & 8 \\ 2 & 0 & 0 & 0 & 1 & 9 \end{pmatrix}$$

$$\begin{pmatrix} 3 & 3 & 2 & 1 & 0 & 5 \\ 0 & 0 & 1 & 3 & 1 & 6 \\ 3 & 1 & 2 & 2 & 3 & 7 \\ 2 & 0 & 0 & 2 & 2 & 8 \\ 2 & 0 & 0 & 0 & 1 & 9 \end{pmatrix} \quad \begin{pmatrix} 3 & 3 & 2 & 1 & 0 & 5 \\ 0 & 0 & 1 & 3 & 1 & 6 \\ 3 & 1 & 2 & 2 & 3 & 7 \\ 2 & 0 & 0 & 2 & 2 & 8 \\ 2 & 0 & 0 & 0 & 1 & 9 \end{pmatrix}$$

$$\begin{pmatrix} 3 & 3 & 2 & 1 & 0 & 5 \\ 0 & 0 & 1 & 3 & 1 & 6 \\ 3 & 1 & 2 & 2 & 3 & 7 \\ 2 & 0 & 0 & 2 & 2 & 8 \\ 2 & 0 & 0 & 0 & 1 & 9 \end{pmatrix} \quad \begin{pmatrix} 3 & 3 & 2 & 1 & 0 & 5 \\ 0 & 0 & 1 & 3 & 1 & 6 \\ 3 & 1 & 2 & 2 & 3 & 7 \\ 2 & 0 & 0 & 2 & 2 & 8 \\ 2 & 0 & 0 & 0 & 1 & 9 \end{pmatrix}$$

Convolution involves performing a dot-product operation between each submatrix and a pre-determined kernel matrix. The kernel matrix is the matrix that slides through every sub-matrix and performs a dot-product operation. The kernel represents some feature in the image that we are trying to recognize. Each entry in the output matrix says something about the similarity between the kernel and the corresponding sub-matrix that was used to compute the dot product. Suppose our kernel matrix is

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

The result of the convolution of A with k would be

$$\begin{bmatrix} A_1.k & A_2.k & A_3.k & A_4.k \\ A_5.k & A_6.k & A_7.k & A_8.k \\ A_9.k & A_{10}.k & A_{11}.k & A_{12}.k \end{bmatrix} = \begin{bmatrix} 12 & 12 & 17.0 & 35 \\ 10.0 & 17.0 & 19.0 & 41 \\ 9.0 & 6.0 & 14.0 & 44 \end{bmatrix}$$

Consider how we got $A_4.k$

$$1*0+0*1+2*5+2*3+1*2+6*0+2*0+1*3+7*2$$

Note that the dot product of a matrix with itself is the square of its magnitude, so values in the matrix output that are close to the magnitude squared of the kernel, indicate that that part of the matrix held some important pattern. This is why convolution is so effective at feature extraction. As we will explain later, convolutional neural networks still have fully-connected layers at the end of the network.

V.CONCLUSION

In conclusion, this paper has provided a comprehensive exploration of the integration of linear algebraic methods in the theory and practice of neural networks. Through the lens of linear algebra, we have gained deeper insights into the inner workings of neural networks, elucidating their structure, dynamics, and optimization. Throughout the discussion, we explored various matrix decompositions such as Singular Value Decomposition (SVD) and Eigenvalue Decomposition, showcasing their utility in model interpretation, regularization, and dimensionality reduction. These techniques have proven invaluable for understanding the underlying structure of neural networks and identifying critical features that contribute to their performance.

REFERENCES

[1] Bamieh, Bassam. (2018). “Discovering transforms: A tutorial on circulant matrices, circular convolution, and the discrete fourier transform.” arXiv preprint arXiv:1805.05533.

[2] Bronstein, M. (2022, January 2). “Deriving convolution from first principles.” Medium. Retrieved March 11, 2022, from https://towardsdatascience.com/deriving-convolution-from-first-principles4ff124888028

[3] Faisal, A. A., Ong, C. S. (2020). “Matrix Approximation.” In “Mathematics for Machine Learning." essay, Cambridge University Press. [4] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. (2016). “Deep learning.” MIT press.

[5] Kalman, D. (1996). “A Singularly Valuable Decomposition: The SVD of a Matrix.” The College Mathematics Journal, 27(1), 2–23.

[6] Lay, D. C., Lay, S. R., McDonald, J. (2022). “Linear algebra and its applications.” Pearson Education Limited.

[7] Murphy, Kevin P. (2012). “Machine Learning: A Probabilistic Perspective.” Cambridge: MIT Press. p. 247.

[8] Nielsen, Michael A. (2015). “Neural networks and deep learning.” Vol. 25. San Francisco, CA, USA: Determination press.

[9] Nicholson, W. K. (2018). “Linear algebra with applications ˙ ’’ Open Textbook Library.

[10] Sharma, Sagar, Simone Sharma, and Anidhya Athaiya. (2017). “Activation functions in neural networks.” towards data science 6.12: 310-316.