Linear Algebra for Deep Learning: Matrix Factorization and a Matrix Perspective on Gradient Descent

Nick Kantack, *Masters student, the University of Virginia*

***Abstract*—Presented is an overview illustrating several benefits of a matrix view of deep learning networks and how applications of linear algebra concepts can aid in the design of more accurate neural networks, faster convergence, and better network generality. Specifically, this work focuses on the application of gradient descent in the deep learning of feed-forward neural networks. Several case studies of optical character recognition networks are presented in detail along with a description of how concepts such as matrix rank, linear independence, matrix factorization, and stability can play a significant role in accelerating the rate at which deep learning networks converge towards an acceptable solution. Several learning methods are compared in an attempt to demonstrate the superiority of certain methods, primarily one which uses a hidden layer to induce non-linearity to overcome low rank in the training data. Comparisons are provided between the performance of networks with different numbers of layers, and commentary is provided on the relevance of each for the letter recognition problem space. Additionally, a brief survey of literature provide a glimpse of how similar matrix views are applicable to more complicated applications of gradient descent, including stochastic and higher order gradient descent.**

***Index Terms*—Deep learning, gradient descent, matrix factorization, neural network, rank, forward propagation**

I. Introduction

DEEP learning has become a cornerstone of artificial intelligence applications over the past decade. Tasks spanning image recognition, game strategy [1], investment planning [2], search results sorting [3], forecasting [4], cancer diagnosis [5], and many other applications have increasingly been handed over to artificial neural networks due to their remarkable effectiveness. With the advent of "big data" as the new norm for data science applications and the proliferation of sensing infrastructure through the Internet of Things, deep learning networks will likely become a increasingly valuable resource in the effort to build intelligent systems that can make effective use of this data abundance. Even the long-used, man-made algorithms of old will increasingly be supplanted or augmented by a wave of better, faster, more agile methods built by machines.

"Deep learning" in the context of this paper is meant to encompass learning as applied to multilayered networks where the human designer prescribes but does not control the learning process. This learning process is typically driven by a cost or loss function that quantifies the error in a network's output and forms the basis for changes to the network that reduce the error and facilitate the learning process. Deep learning can take many forms, but this paper will focus specifically on deep learning in the context of artificial neural networks (ANNs).

Artificial neural networks are a (usually simulated) network of discrete elements known as neurons [6] [7]. Each neuron is connected to other neurons through synapses. Each neuron can have multiple inputs and multiple outputs, but traditionally only a single state, often a numerical value referred to as the neuron's *activation* which is a function of its inputs. The activations of neurons affect the activations of others through the synapses. Each synapse is a one-way connection between two neurons and has a weight associated with it which indicates how strongly the activation of the neuron on the input end of the synapse can influence the activation of the neuron on the output end of the synapse.

While there several types of ANNs, each consists of the same primitive components. A great deal of research over the past 70 years has poured into understanding, characterizing, and optimizing the performance of ANNs, along with efforts to design hardware, software, and computing paradigms that lead to faster, smarter, and more versatile networks.

A. Gradient descent and the learning process

As described, ANNs may sound like rudimentary (and slow) software abstractions of electrical circuits. The essence of an ANN's utility lies in its ability to learn with time. While physical circuits do not rewire themselves to make improvements, an ANN is able to evolve in response to the data it encounters, to learn from its mistakes, and to rapidly improve in its ability to accomplish a specified task.

The most common learning method is *gradient descent* [7]. Gradient descent refers to the process of calculating the gradient of a loss function with respect to the network's parameters, and then making small changes in those parameters to reduce the loss function. Gradient descent can be visualized as a journey in the parameter-space of the network with the goal of descending the loss function to a destination that is a minimum of the loss function. Gradient descent is fairly simple on surface, but performing gradient descent well or optimally is much more nuanced. Taking steps that are too large can cause a network to overshoot the destination or enter a feedback loop that diverges away from the desired performance. Taking steps that are too small can lead to prohibitively long learning times and/or high computational costs to learn. Therefore, a disciplined approach rooted in foundational linear algebra

This work was completed as a term paper for ECE 695: Topics in Linear Algebra administered at Old Dominion University in the summer of 2020.

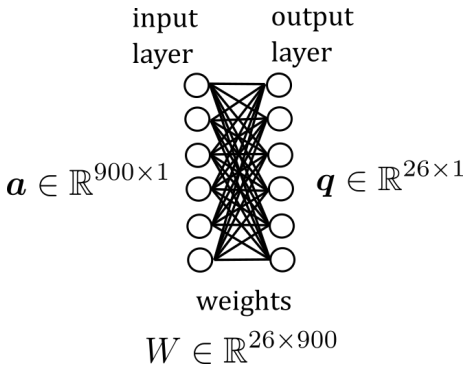

Fig. 1. A schematic for a simple two layer feed forward neural network used in the first case study of this paper.

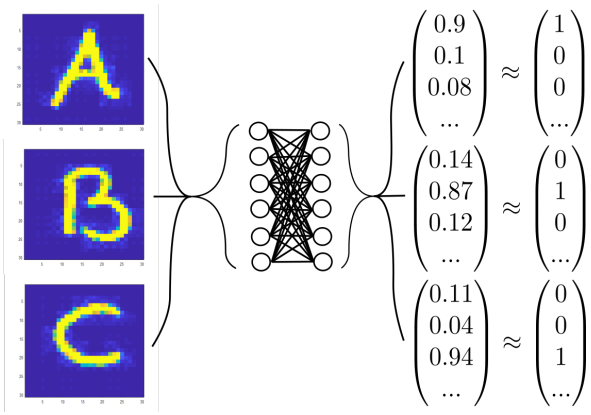

Fig. 2. Raw pixel data from hand written letters become the activations for the input layer of the network. The network produces a vector with a maximal element, and the maximal element's index corresponds to the index of the letter that is the network's classification of the image.

concepts can mean the difference between a successful and unsuccessful network.

II. Gradient Descent in ANNs

We will now consider a simple artificial neuron network and illustrate how a matrix formalism can provide insights when designing such networks. The formalism developed in this paper is very similar to that used in other works concerning feed-forward neural networks [6].

Suppose that we have an artificial neural network with an input and output layer. Let us represent the states, or *activations*, of the input neurons in a vector $\boldsymbol{a}$ and the activations of the output neurons in a vector $\boldsymbol{q}$. We can define a matrix W to represent the weights of each synapse from a neuron in $\boldsymbol{a}$ to a neuron in $\boldsymbol{q}$ (W_{ij} is the weight of neuron a_j on neuron q_i). The purpose of this network is to examine a 30 pixel by 30 pixel image of a handwritten capital English letter (e.g. "A," "B," etc.) and to determine which of the 26 letters is in the image (this task is referred to as optical character recognition, and is described in [7]). Thus, we will make an input layer with 900 neurons (one neuron per pixel). The output layer will contain 26 neurons, each neuron represents a letter in the alphabet. We thus have the sizes

$$\begin{aligned} \boldsymbol{a} \in \mathbb{R}^{900\times 1} &\quad - \quad \text{input activations} \\ \boldsymbol{q} \in \mathbb{R}^{26\times 1} &\quad - \quad \text{output activations} \\ W \in \mathbb{R}^{900\times 26} &\quad - \quad \text{input weights} \end{aligned} \tag{1}$$

Figure 1 gives a concise overview of the network. It is helpful to depict the forward propagation process as a matrix operation. The output activations are related to the input activations in the following way.

$$\boldsymbol{q} = W\boldsymbol{a} \tag{2}$$

The result is a vector in $\mathbb{R}^{26\times 1}$, and we will take the index of the largest element to correspond to the network's best guess for the letter represented by the image whose pixels created the input activations $\boldsymbol{a}$. In this scheme, we could claim that a perfect identification on the part of the network would consist of the output activations being equal to a canonical unit vector corresponding to the correct letter. Let us define a set of correct output vectors $\boldsymbol{c}_A$, $\boldsymbol{c}_B$, $\boldsymbol{c}_C$, etc.

$$\boldsymbol{c}_A = \begin{pmatrix}1\\0\\0\\\dots\end{pmatrix} \quad \boldsymbol{c}_B = \begin{pmatrix}0\\1\\0\\\dots\end{pmatrix} \quad \boldsymbol{c}_Z = \begin{pmatrix}\dots\\0\\0\\1\end{pmatrix} \tag{3}$$

We can now define the loss function for this network. Suppose that the network is analyzing an image of the letter "A." We can define the loss function as the squared error of the output.

$$\mathcal{L}(\boldsymbol{q}) = ||\boldsymbol{q} - \boldsymbol{c}_A||^2 \tag{4}$$

We wish to apply gradient descent to improve the network, so we need to determine the gradient of the loss function with respect to each weight. To do so we'll observer the following.

$$\frac{\partial \mathcal{L}}{\partial W_{ij}} = 2(q_i - c_{A,i})a_j \tag{5}$$

From which we can define the learning step. Let us define a learning rate constant $\alpha > 0$. Then we can update the weights in the following way.

$$W \leftarrow W - 2\alpha(\boldsymbol{q} - \boldsymbol{c}_A)\boldsymbol{a}^T \tag{6}$$

A. Recasting as a Factorization Problem

Before we begin training the network on example data, it is valuable to examine the weight optimization as a matrix factorization problem. If we created an alphabet matrix A whose columns encoded prototypical letters, then our desired output would be the identity matrix in $\mathbb{R}^{26\times 26}$.

$$\begin{pmatrix}\boldsymbol{c}_A & \boldsymbol{c}_B & \boldsymbol{c}_C & \dots\end{pmatrix} = I^{26\times 26} = \widetilde{W}A \tag{7}$$

In this case, the perfect set of weights $\widetilde{W}$ is the left inverse of A.

$$\widetilde{W} = (A^TA)^{-1}A^T \tag{8}$$

A discussion of rank is needed here. We have assumed that $\text{rank}(A) = 26$. If $\text{rank}(A) < 26$, then A^TA is not invertible. $\text{rank}(A) = 26$ would imply that the prototypical letters are

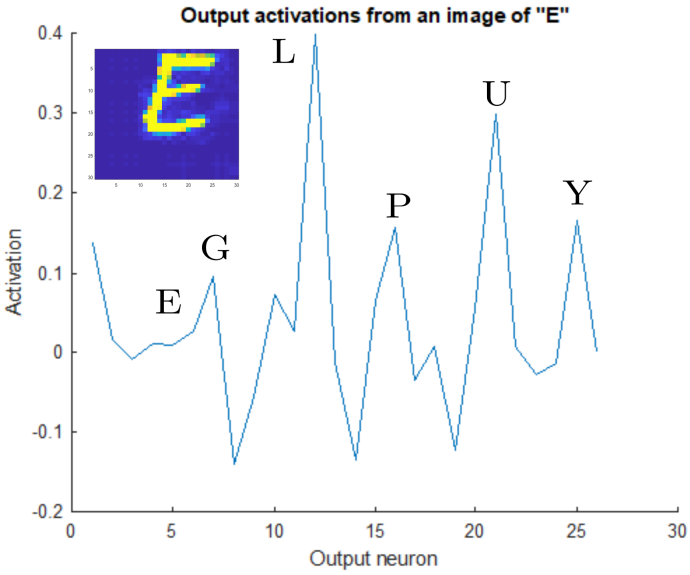

Fig. 3. The network output for an input image of an "E" using weights that were obtained from (8). Note that the network has incorrectly classified this "E" as an "L," a consequence of the low rank nature of the A matrix used to generate the weight matrix $\widetilde{W}$. The prevalence of other non-"E" spikes in the output activations hints at many linear combinations of letters that occupy the null space of A.

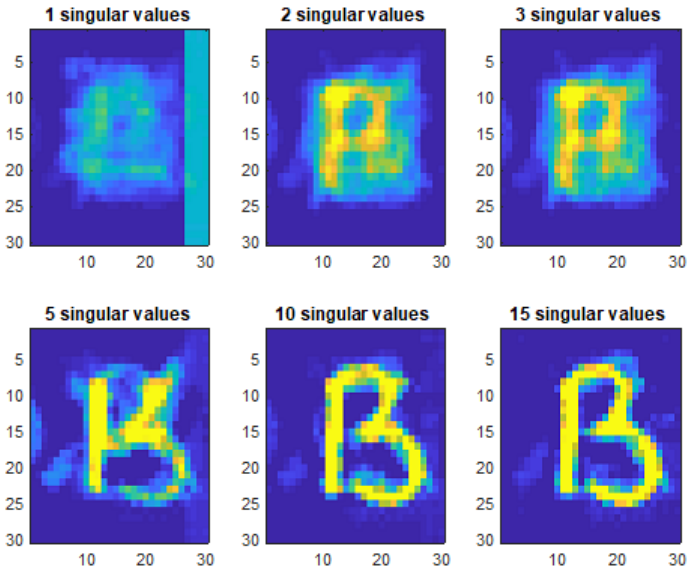

Fig. 4. The low rank approximation $U(:, 1:r)\Sigma(1:r, 1:r)V^T(1:r, :)$ for the prototypical letter "B" is show for various numbers of singular values r. Note that with as few as 5 singular values used, the letter "B" becomes distinguishable. This illustrates that the alphabet matrix A has an effective rank close to 5, and thus the effective nullspace of A could have dimension as high as 21. This indicates that there are many linear combinations of letters that can be mistaken for other letters.

linearly independent, i.e. that no letter in the English language can be accurately factored as a linear combination of other letters. This assumption turns out to be a poor one, i.e. there *are* linear combinations of English letters that produce others. Some examples are presented below.

$$L + F \approx E \tag{9a}$$

$$L + T - B + 3 \approx I \quad \text{if numbers are included} \tag{9b}$$

These combinations of equations would imply the following.

$$\boldsymbol{c}_E = \widetilde{W}\boldsymbol{a}_E \approx \widetilde{W}(\boldsymbol{a}_L + \boldsymbol{a}_F) \tag{10}$$

$$\boldsymbol{c}_E \approx \widetilde{W}\boldsymbol{a}_L + \widetilde{W}\boldsymbol{a}_F = \boldsymbol{c}_L + \boldsymbol{c}_F \tag{11}$$

This is, obviously, a contradiction (the vectors $\boldsymbol{c}$ are linearly independent by design), which hints at the unattainability of such a perfect set of weights $\widetilde{W}$. Even if we add a bias vector $\boldsymbol{b}$ to the outputs ($\boldsymbol{q} = W\boldsymbol{a} + \boldsymbol{b}$), we find

$$\boldsymbol{c}_E = \boldsymbol{c}_L + \boldsymbol{c}_F - \boldsymbol{b} \tag{12}$$

where $\boldsymbol{b}$ is only available to compensate this linear combination (so $\boldsymbol{b}$ will not resolve other linear combinations of letters). Thus, we have encountered a fundamental limitation in the two layer network's ability to learn a data set with a multidimensional dimensional effective null space.

B. Assessing the effective rank of A

We take a brief detour to examine the rank of the training data to highlight the impact it has on the network's performance. Taking the singular value decomposition (SVD) of our alphabet matrix A will facilitate an analysis of effective rank. The singular values of A will reveal information about the effective rank of our data.

The singular value decomposition of A yields a matrix, U, of left singular vectors, a matrix, Σ, of singular values, and a matrix, V, of right singular vectors. In this case, $A = U\Sigma V^T$. If A is full rank (in our case, $\text{rank}(A) = 26$) then the main diagonal of Σ will contain 26 non-zero elements. If A is rank $n < 26$, then the diagonal of Σ will have n non-zero elements. If A is *approximately* rank $n < 26$ (that is, $\text{rank}(A) = 26$, but $A \approx \widetilde{A}$ where $\text{rank}(\widetilde{A}) = n$), then the main diagonal of Σ will contain n elements that are significantly greater than zero, and $26 - n$ elements that are approximately zero. If we seek to approximate A as a matrix of rank $r < 26$, the best choice would be the matrix obtained by the outer product of the first r left and right singular vectors of A scaled by the first r singular values.

$$A \approx U(:, 1:r)\Sigma(1:r, 1:r)V^T(1:r, :) \tag{13}$$

Figure 4 shows the visual quality of several low rank approximations for A, which illustrated how the effective rank of A is significantly less than 26.

Figure 5 shows the singular values of A plotted on a single figure. The size of the first two singular values relative to the rest indicates that the images of the letters are significantly well approximated by the first two singular vectors (see Figure 4 for "2 singular values," and note how the letter "B" is almost distinguishable). This indicates that while $\text{rank}(A) = 26$, A can be very accurately approximated by a much lower rank matrix. This means that the effective null space of A (a pseudo-subspace of vectors that produce insignificant changes in A) has considerably high dimension, and therefore

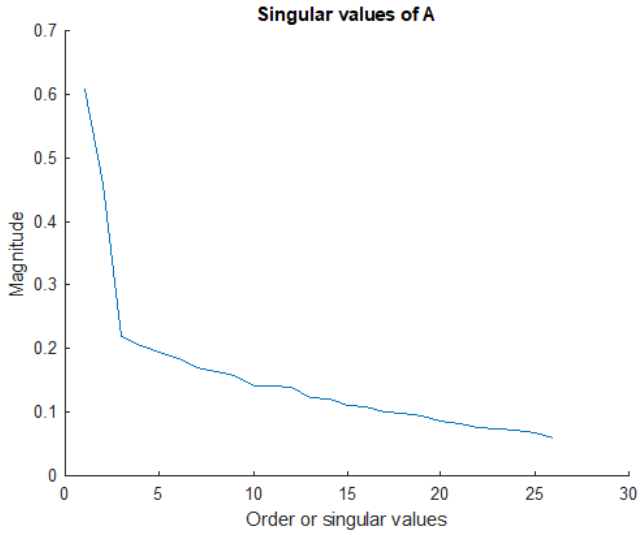

Fig. 5. The singular values of the alphabet matrix A are plotted as a function of their position in the diagonal of Σ. Singular values are always decreasing. These singular values indicate that while $\text{rank}(A) = 26$, A is well approximated by a much lower rank matrix, and thus many linear combinations of letters produce equivalent images.

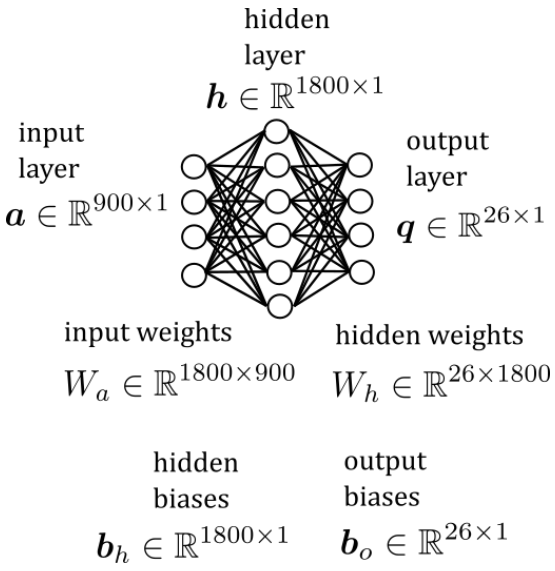

Fig. 6. A schematic for a feed forward neural network with a single hidden layer. Note that there are now two weight matrices as well as additional bias matrices.

there are many different ways to take linear combinations of letters and produce extremely similar images. As we've demonstrated, this produces a challenge for our two layer network since letters can be mistaken for linear combinations of others, leading to inaccurate network performance. One way of mitigating the impact of the low rank of our data on the network's learning process is the addition of a *hidden* layer in the network between the input and output layers. Adding a hidden layer to such a neural network can significantly raise the flexibility (and ultimate accuracy) of the solution, but at the obvious cost of computational complexity. Jain [7] provides a discussion on the capabilities of two layer networks vs. three layer networks. In our case, we will find that the addition of the hidden layer induces non-linearity in the transformation from image space to letter space which allows the network to develop a resistance to the low rank nature of the training data. Details of the three layer network employed in our case study will now be presented.

III. Adding a hidden layer

A second neural network is now introduced. This network contains an additional, "hidden" layer whose activations are strictly determined by the input layer activations and whose activations strictly determine the output layer activations (along with biases). There is a matrix of weights for the synapses from the input layer to the hidden layer and a separate matrix of weights for the synapses from the hidden layer to the output layer. Finally, the neurons in the hidden layer and the output layer each have their own biases, which are constants added to the activation prior to thresholding. These biases are learned along with the weights during the training process. The format for supplying input images to the network as well as the format for judging the network's letter classification from the output is the same as in the case of the two layer network. We will define the following notation.

$$\begin{array}{ccc} \boldsymbol{a} \in \mathbb{R}^{900\times 1} & - & \text{input activations} \\ \boldsymbol{h} \in \mathbb{R}^{1800\times 1} & - & \text{hidden activations} \\ \boldsymbol{q} \in \mathbb{R}^{26\times 1} & - & \text{output activations} \\ W_a \in \mathbb{R}^{1800\times 900} & - & \text{input weights} \\ W_h \in \mathbb{R}^{26\times 1800} & - & \text{hidden weights} \\ \boldsymbol{b}_h \in \mathbb{R}^{1800\times 1} & - & \text{hidden biases} \\ \boldsymbol{b}_q \in \mathbb{R}^{26\times 1} & - & \text{output biases} \end{array} \tag{14}$$

Let us employ the following threshold function, $\sigma(\boldsymbol{x})$.

$$\sigma(\boldsymbol{x}) \in \mathbb{R}^{\text{rows}(\boldsymbol{x})\times 1} \quad \sigma(\boldsymbol{x})_i = \frac{1}{1+e^{-x_i/150}} \tag{15}$$

The purpose of the thresholding function is to avoid divergence in the size of the weights [7]. The factor of 1/150 in the sigmoid function is chosen to be a few times larger than 1/900, the reciprocal of the number of input neurons. This prevents the activations from saturating too quickly. Jain [2] provides commentary on the process of determining a suitable denominator in the exponential of the thresholding function. While some rules of thumb exist, a parameter like this is often determined empirically through trial and error.

Forward propoagation through the network yields the following:

$$\boldsymbol{q} = \sigma(W_h\boldsymbol{h} + \boldsymbol{b}_h) = \sigma(W_h\sigma(W_a\boldsymbol{a} + \boldsymbol{b}_q) + \boldsymbol{b}_h) \tag{16}$$

Updating the weights is a more complicated endeavor in this new network and will required the use of *backpropagation*. As before, we can ascribe an error (e.g. in the case of "A") as $||\boldsymbol{q} - \boldsymbol{c}_A||^2$. This allows for an update step for W_h similar to (5).

$$W_h \leftarrow W_h - 2\alpha(\boldsymbol{q} - \boldsymbol{c}_A)\boldsymbol{h}^T \tag{17}$$

To update W_a, we need to attribute an error vector, $\boldsymbol{v}$ to the hidden layer which can serve as the basis for a similar weight update. We will use the following.

$$\boldsymbol{v} = W_h^T(\boldsymbol{q} - \boldsymbol{c})\sigma'(\widetilde{\boldsymbol{h}}) \tag{18}$$

$\widetilde{\boldsymbol{h}}$ is the vector of activations of the hidden layer prior to applying the threshold function (15). It is worth recognizing here that this relation admits a recursive form that can easily extend to networks with multiple hidden layers $\left(\boldsymbol{v}_k = W_h^T \boldsymbol{v}_{k+1}\sigma'(\widetilde{\boldsymbol{h}}_k)\right)$.

The learning process evolves via traditional backpropagation. Each non-input layer is ascribed an error vector, $\boldsymbol{v}$, from which we can compute the updates to the weights and biases. Let the "correct" output, c, for a given letter be the vector in $\mathbb{R}^{26\times 1}$ with all zero elements except for a value of 1 for the element corresponding to the correct letter.

$$\boldsymbol{v}_q = \boldsymbol{q} - \boldsymbol{c} \qquad \boldsymbol{v}_h = W_h^T \boldsymbol{v}_q \tag{19}$$

And finally we can define the updates to the weights and biases during the learning process for a learning rate α (in this experiment, $\alpha = 0.02$).

$$\begin{aligned} W_h &\leftarrow W_h - \alpha \boldsymbol{v}_h \boldsymbol{h}^T \\ W_q &\leftarrow W_q - \alpha \boldsymbol{v}_q \boldsymbol{q}^T \\ \boldsymbol{b}_h &\leftarrow \boldsymbol{b}_h - \alpha \boldsymbol{v}_h \\ \boldsymbol{b}_q &\leftarrow \boldsymbol{b}_q - \alpha \boldsymbol{v}_q \end{aligned} \tag{20}$$

A. Hidden layer impact on low rank data

It is worth addressing explicitly how the addition of the hidden layer helps overcome the impact of low rank data on the network's convergence rate. We will ignore the sigmoid function since its effect is linear for most neurons. Suppose, for illustration purposes, that

$$\boldsymbol{a}_E = \boldsymbol{a}_F + \boldsymbol{a}_L \tag{21}$$

Premultiplying by W yeilds the contradiction

$$\boldsymbol{c}_E = \boldsymbol{c}_F + \boldsymbol{c}_L \tag{22}$$

The important conclusion is that *no linear mapping exists that can avoid this contradiction*. This is a fundamental shortfall of the two layer network. Even with the addition of a bias layer $\boldsymbol{b}$, the result

$$\boldsymbol{c}_E = \boldsymbol{c}_F + \boldsymbol{c}_L - \boldsymbol{b} \tag{23}$$

overburdens the bias vector $\boldsymbol{b}$ to compensate for the $\boldsymbol{a}_E = \boldsymbol{a}_F + \boldsymbol{a}_L$ case ($\boldsymbol{b}$ is not available to compensate any other superposition of letters).

Now we examine forward propagation with a hidden layer. In this simplified case,

$$\boldsymbol{q} = W_q(W_h\boldsymbol{a} + \boldsymbol{b}_h) + \boldsymbol{b}_q \tag{24}$$

Then for the case $\boldsymbol{a}_E = \boldsymbol{a}_F + \boldsymbol{a}_L$ we find

$$\boldsymbol{c}_E = W_q(W_h(\boldsymbol{a}_F + \boldsymbol{a}_L) + \boldsymbol{b}_h) + \boldsymbol{b}_q \tag{25}$$

$$\boldsymbol{c}_E = W_q(W_h\boldsymbol{a}_F + \boldsymbol{b}_h) + W_h\boldsymbol{a}_L + \boldsymbol{b}_q \tag{26}$$

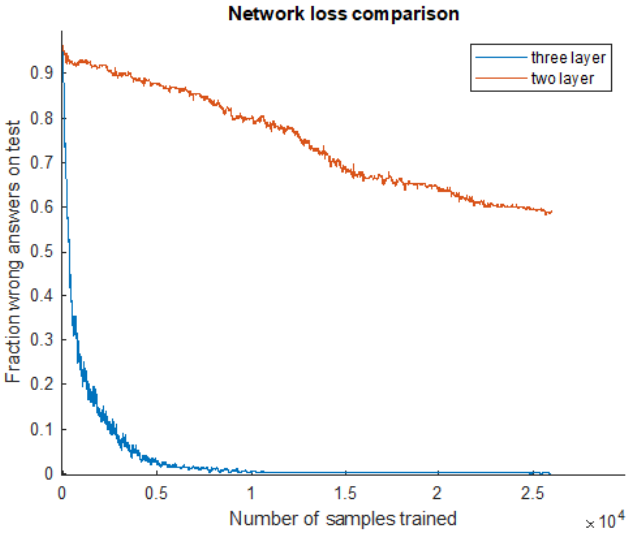

Fig. 7. A comparison of the evolution of error across the two discussed neural networks as a function of the number of training samples learned. The test consists of asking the networks to classify 520 hand written letters. Recorded is the fraction of those 520 letters that were incorrectly classified. This test was performed after each training sample was learned, and the data for the test were pulled from the same dataset as the sample data.

$$\boldsymbol{c}_E = \boldsymbol{c}_F + \boldsymbol{c}_L - W_q\boldsymbol{b}_h - \boldsymbol{b}_q \tag{27}$$

In this case, W_q can take on values to make the above expression true. Additionally, W_q can be decoupled from W_h by making an adjustment V to W_q in the following way.

$$W_q \leftarrow W_q V \tag{28a}$$

$$W_h \leftarrow V^T(VV^T)^{-1}W_h \tag{28b}$$

In this way, the three layer network has unique flexibility to provide convergence despite the low rank nature of the sample space.

One might wonder whether the improvement in Figure 7 would continue with the addition of a second hidden layer. Our motivation for adding the first hidden layer was to reduce the impact of the null space of the training data on the network's performance. To illustrate, and network was constructed with an additional hidden layer of 1800 neurons (other dimensions remained the same as in the three layer network from before). The convergence of the four layer network is noticeably slower than that of the three layer network (Figure 8), as well as the stability of the network's performance over the course of the training process.

Perhaps the main difference between the networks which manifests in the dissimilar convergence rates is the significantly different number of weights. The three layer network has approximately 1.7 million weights, while the four layer network has approximately 4.9 million weights. Therefore, the configuration space of the four layer network is considerably bigger, and traversing towards a local minimum in the cost function of such a higher dimensional parameter space takes significantly longer. Additionally, adjustments to weights in the first two layers is has a magnified effect as those changes are propagated through the subsequent weight matrices. This

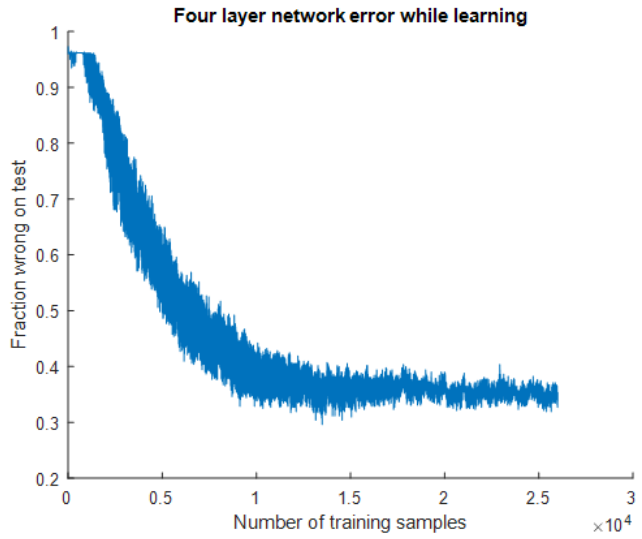

Fig. 8. Network test performance over time for a four layer neural network. Note that the convergence rate is slower than that of the three layer network, and the stability of the network's performance is much lower. This poorer performance is largely due to the addition of unnecessary degrees of freedom in the expanded network.

leads to the network performance becoming more unstable during the learning process.

The parameters of the gradient descent process (the learning rate and the sigmoid function) in this case study were selected empirically from multiple trials to produce better convergence rates. Due to the complexity of networks like the three layer network, the optimal values for these parameters can be difficult to determine analytically and must often be chosen from empirical data [7]. However, in the case of certain simple networks, matrix factorization can yield analytically optimal learning rates which can form the basis for designing learning processes for simple and complex networks. A discussion on one such analytic solution is not presented.

B. Eigenvalue decomposition of optimal learning rate

It is useful to select an appropriate learning rate. Ideally, one would use a learning rate that was optimal (leading to the best possible convergence to an acceptable solution for a given number of training samples). Incidentally, a matrix view of our network can provide some insight on an appropriate learning rate.

Cun [6] examined a two layer network with a single output neuron and analytically arrived at optimal learning rate parameters. Specifically, they find that the optimal learning rate is obtained analytically from the eigenvalue decomposition of a covariance matrix R which describes the covariance of training samples. In our notation,

$$R_{ij} = \frac{1}{26}\sum_{k=1}^{26} A_{ik}A_{jk} \quad R \in \mathbb{R}^{900\times 26}. \tag{29}$$

R is the Hessian matrix of our loss function. If we define the vector $\boldsymbol{z}$ as

$$z_i = \frac{1}{p}\sum_{k=1}^{26} \boldsymbol{c}_i^T \boldsymbol{a}_k \quad \boldsymbol{z} \in \mathbb{R}^{900\times 1} \tag{30}$$

Then the weight optimization emerges from solving the linear system

$$RW = \boldsymbol{z} \tag{31}$$

and the optimal learning rate emerges from the eigenvalues of the covariance matrix, R. Specifically, the optimal learning rate is given by $\alpha = 1/\lambda_{\text{max}}$ where λ_{max} is the largest eigenvalue of R. This optimization chooses the largest learning rate that is still guaranteed to converge towards a solution in the weight update recursion.

IV. Beyond vanilla gradient descent

The simple case of gradient descent that was presented in the previous case studies is sometimes referred to as "vanilla gradient descent" and represents a basic application of the learning method. Here we present a brief discussion on more complicated implementations of gradient descent and their respective strengths. As Cun [6] points out, it is difficult to make the case that one implementation of gradient descent is superior to others. Rather, the greatest success often follows from selecting the most appropriate version of gradient descent for a particular problem.

A. Stochastic Gradient Descent

Stochastic gradient descent performs the same update on any given weight as would be performed by vanilla gradient descent, but stochastic gradient descent only updates a randomly selected subset of the weights and biases on any given training sample. Bottou [8] provides a comparison of the theoretical convergence times for vanilla gradient descent vs stochastic gradient descent and demonstrates how the time to convergence can be significantly shorter utilizing the stochastic method.

B. Higher order gradient descent

In our case studies, the gradient descent process factored in only first order information (weight updates were of the form $-\alpha(\boldsymbol{a} - \boldsymbol{c})$ where α was a positive constant). Bottou [8] provides a discussion of *second order gradient descent* which factors in the curvature of the loss function to provide a higher convergence rate. In particular, α is replaced with an inverse Hessian matrix of the loss function. This requires the additional computation of the inverse Hessian in order to perform an update to the weights, but the overall convergence time might be reduced if the cost function is sufficiently smooth. Schraudolph [9] identifies several matrix based methods for incorporating loss function curvature into the weight optimization process. In fact, as Bottou [8] points out, second order gradient descent can reach the optimal solution after a single step in the special case that the loss function is quadratic.

C. Learned gradient descent

Andrychowicz [10] demonstrates an extension of gradient descent where the optimization process itself is subject to learning and updating as the network trains. In this case, the optimization function itself is parametrized, and these parameters themselves induce a gradient on the loss function that allows gradient descent learning to apply to the parameters of the function that are fed into the gradient descent learning process for the neurons. The advantage of this additional use of gradient descent on the learning process is to free up the network to learn more appropriate parameters for the gradient descent of its weights than those that a human designer would have otherwise rigidly enforced. Ionescu [11] demonstrated a matrix formalism for estimating gradients for backpropagation in non-linear networks. Depending on the nature of the network's ultimate task, the learned gradient descent parameters can provide significantly faster convergence towards a solution.

V. Conclusion

The examples considered illustrate how a matrix perspective of deep learning networks plays an important role in their design and optimization. Eigenvalue decomposition and stability analysis are useful for selecting learning rates which are as large as possible without leading to divergence away from the solution. Considerations with regard to matrix rank and nullspaces provide insight into whether or not a particular network will be able to coverge towards a correct solution with a particular training data set. We've also illustrated how these basic concepts extend toward more complicated networks, indicating that disciplined applications of linear algebra concepts will continue to be a foundational part of the most successful deep learning networks.

Acknowledgment

I thank Dr. Dimitrie Popescu for his assistance throughout the course for which this paper was created. This work was facilitated through an online linear algebra course (ECE695) administered by Old Dominion University in Norfolk, Virginia.

References

[1] D. Silver, J. Schrittwieser, et al. "Mastering the game of Go without human knowledge," *Nature*. 550, 354-359 2017

[2] C. L. Sabharwal, "The rise of machine learning and robo-advisors in banking," *IJBT* 2:28-34 (2018).

[3] P. Huang, X. He, J. Gao, L. Deng, A. Acero, L. Heck, "Learning deep structured semantic models for web search unsing clickthrough data," Proceedings of the 22nd ACM international conference on Information and Knowledge Management. 2333-2338 (2018) DOI:10.1145/2505515.2505665

[4] A. G. Salma, B. Kanigoro, Y. Heryadi, et al. "Weather forecasting using deep learning techniques," 2015 International Conference on Advanced Computer Science and Information Systems (ICACSIS), Depok, 2015, pp. 281-285, doi: 10.1109/ICACSIS.2015.7415154.

[5] S. M. McKinney, M. Sieniek, V.Godbole, et al. "International evaluation of an AI system for breast cancer screening," *Nature*. 577, 89-94 January 2020.

[6] Y. L. Cun, I. Kanter, S. A. Solla, "Eigenvalues of Covariance Matrices: Application to Neural-Network Learning," in *Physical Review Letters* vol. 66, no. 18, The American Physical Society, 1991, pp. 2396–2399.

[7] A. K. Jain, J. Mao, K. M. Mohiuddin, "Artificial Neural Networks: A Tutorial," Computer, vol. 29, no. 3, pp. 31-44, March 1996, doi: 10.1109/2.485891.

[8] L. Bottou, "Large-Scale Machine Learning with Stochastic Gradient Descent," Proceedings of COMPSTAT 2010 DOI 10.1007/978-3-7908-2604-3 16, Springer-Verlag Berlin Heidelberg 2010

[9] N. N. Schraudolph, "Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent," MITP, Neural Computation 2002, vol. 14, issue 7. DOI: 10.1162/08997660260028683. ISSN: 0899-7667

[10] M. Andrychowicz, M. Denil, S. G. Colmenarejo, M. W. Hoffman, D. Pfau, T. Schaul. Brendan Schillingford, N. de Freitas, "Learning to learn by gradient descent by gradient descent" 30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

[11] C. Ionescu, O. Vantzos, C. Sminchisescu, "Matrix Backpropagation for Deep Networks with Structured Layers," ICCV 2015, Computer Vision Foundation. p. 2965–2973