Computational complexity reduction of deep neural networks

Mee Seong Im
Department of Mathematics
United States Naval Academy
Annapolis, MD 21402
meeseongim@gmail.com

Venkat Dasari
U.S. Army Combat Capabilities Development Command
U.S. Army Research Laboratory
Aberdeen Proving Ground, MD 21005
venkateswara.r.dasari.civ@mail.mil

***Abstract*—Deep neural networks (DNN) have been widely used and play a major role in the field of computer vision and autonomous navigation. However, these DNNs are computationally complex and their deployment over resource-constrained platforms is difficult without additional optimizations and customization.**

In this manuscript, we describe an overview of DNN architecture and propose methods to reduce computational complexity in order to accelerate training and inference speeds to fit them on edge computing platforms with low computational resources.

***Index Terms*—Multilayer models, machine learning, neural network, computational complexity, computation reduction.**

I. Introduction

Deep neural network design problems support both theoretical and experimental frameworks of machine learning [1], [2], [7], [8], [14]. They can detect objects, recognize letters, digits, and symbols, and process scenarios such as running children and a crying woman. They are used in video and monitoring systems to understand established social and power networks via marriage alliance and business relations, to model complex scientific collaborations, to examine multilayered climate dynamics and chain reaction of behavior of atoms in molecular biology. They may be used to represent air transportation networks for various companies, in which multilayer structure depicts flight connections operated by different airline companies. Complex algorithms allow such models to compute and analyze situations, understand scenarios, and react to their surroundings, distinguishing between friendly and hostile environments.

Multilayer machine learning models have become more complex, with millions of parameters and weights, in order to achieve more accurate and sophisticated outcomes. But all of such operations are not necessary in order to achieve a reasonable output. In fact, they often drain finite computational time and energy resources, not being able to finish the computation within a reasonable time allowed. See, *e.g.*, other research papers that have investigated optimization problems with low tactical computing platforms ([3], [4], [9]–[11]). Therefore, there is an immense problem due to resource constraints on the tactical weights.

The aim of this manuscript is to reduce multilayer machine learning model complexity whilst keep accuracy of the model by at least a reasonable amount, say $(100-\varepsilon)\%$, where ε is small, *e.g.*, 2%. That is, we modify the model in neural networks. The goal is to shrink the architectural aspects of the model so that speed, accuracy, and data are preserved. The unnecessary mathematics and processes are removed, depending on the computation, and we execute this by reducing certain computational aspects of model layers in a systematic way.

II. Neural network structures

A multilayer machine learning model is an artificial neural network, which is composed of multiple layers of nodes with threshold activation [1], [2], [7], [8], [14]. It consists of an input layer, (multiple) hidden layers, and an output layer. The nodes are weighted, with higher weights representing activated neurons. See Figure II.1.

There are several primary neural network architectural structures. Convolutional neural networks are primarily designed for image recognition, recurrent neural networks best produce predictive results in sequential data, and artificial neural networks model

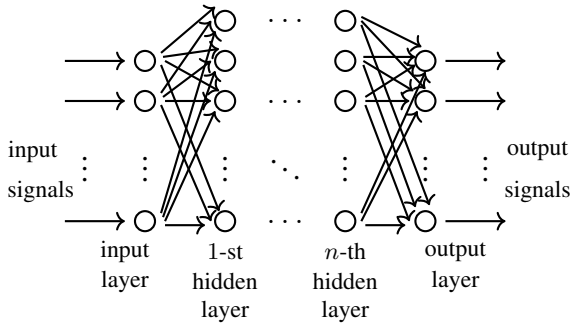

Fig. II.1. A directed graph of a multilayer machine learning model with connections and weights. It represents a feedforward network with $n + 2$ layers, input layer, output layer, and n hidden layers.

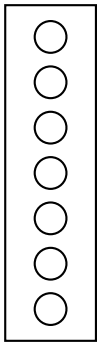

Fig. II.2. The only structural layer in the entire network.

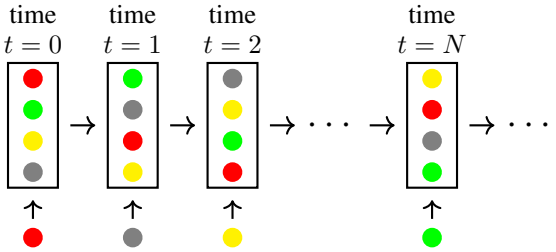

Fig. II.3. An example of the structure of recurrent neural network, which can be thought of as a passage through time. There are 4 time stamps at $t = 0$. At $t = 1$, the net takes the output of time $t = 0$ and send it back into the net along with the next input. This procedure is repeated in the net.

nonlinear problems to predict the output values for given input parameters from their training values.

A. Convolutional neural networks

In convolutional neural networks, images of a letter, number, or a symbol at a low resolution can be identified by a deep neural network. For example, if the image of a single digit is on a grid of 28 by 28 pixels, then the input consists of 28^2 input data while the output consists of a single number between 0 to 9. There are multiple hidden layers, with signals forming a feedforward neural network, flowing only in one direction from input to output, see, *e.g.*, Figure II.1.

B. Recurrent neural networks

If patterns in a data set change with respect to time, then recurrent neural networks would best represent such data set. This machine learning model has a structure with a built-in feedback loop, allowing it to act as a forecasting engine. They are applied from speech recognition to driverless vehicles and robotics. Figure II.2 shows the one and only structural layer in the entire network. But here, the output of a layer is added to the next input and fed back into the same layer, which is usually the only layer in the entire network. This process could be thought of as a passage through time. See Figure II.3. Also see [5], [6], [12], [13] for more detail.

Unlike feedforward nets, a recurrent neural network receives a sequence of values as input, as well as produces a sequence of values as output. Its ability to operate with sequences opens up these nets to a wide variety of applications. For example, with a single input and a sequence of outputs, one such application is image captioning. On the other hand, a sequence of inputs with a single output may be used for document classification. When both the input and the output are sequences, these nets can classify videos frame by frame. If a time delay is introduced, the neuron network can statistically forecast that demand in supply chain planning.

C. Artificial neural network

Artificial neural networks are computing systems designed to simulate by analyzing and processing information. As the human brain has around 1.0×10^{12} neurons, artificial neural networks are designed in programmable machines to behave like interconnected brain cells.

Each connection can transmit a signal to other neurons. An artificial neuron that receives a signal then processes it and can signal neurons connected to it. The signal at a connection is a real number, and the output of each neuron is computed by some nonlinear function of the sum of its inputs.

III. Neural network learning

Neurons are connected to each other in various patterns, to allow the output of some neurons to become the input of others; they form a directed, weighted graph.

A. Neurons

Each artificial neuron has inputs and produces a single output which can be sent to multiple other neurons. Initial inputs may be the feature values of some external data, such as images or documents, or they can be the outputs of other neurons. The outputs of the final output neurons of the neural net accomplish the task, such as recognizing an object in an image or letters and symbols.

We follow the following procedure to obtain the output of the neuron. First, we take the weighted sum of all the inputs, weighted by the weights of the connections from the inputs to the neuron. We add a bias term to this sum. This weighted sum is also called the activation. This weighted sum is then passed through a (usually nonlinear) activation function to produce the output. The ultimate outputs accomplish the task.

B. Connections and weights

The edges are called connections. Neurons and connections have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons have a threshold such that a signal is sent only if the aggregate signal crosses that threshold.

Neurons are aggregated into layers, with different layers performing different transformations on their inputs. Signals travel from the first layer, also known as the input layer, to the last layer, also known as the output layer, possibly after traversing the layers multiple times.

C. Hidden layers

In multilayer neural networks, hidden layers are located between the input and the output layer of the algorithm. The function programmed into the network applies weights to the inputs, adds appropriate bias, and directs them through an activation function as the output. They perform nonlinear transformations of the inputs entered into the network since they are designed to produce an output specific for an intended result. Often times, they are useful when the intended output of the algorithm is a probability, since they take an input and produce an output value x, where $0 \leq x \leq 1$.

Hidden layers allow for the function of a neural network to be broken down into specific transformations of the data, where each hidden layer function is specialized to produce a defined output. As an example, a hidden layer that is used to identify ears and eyes cannot solely identify the person. However, when placed in conjunction with additional hidden layers used to identify the facial structure, hair, body type, etc., the neural network can then make predictions and identify the correct individual within visual data.

Figure III.1 is an example of a simple neural network, which has one hidden layer, while Figure II.1 is an example of a more general neural network, with $n \geq 1$ hidden layers.

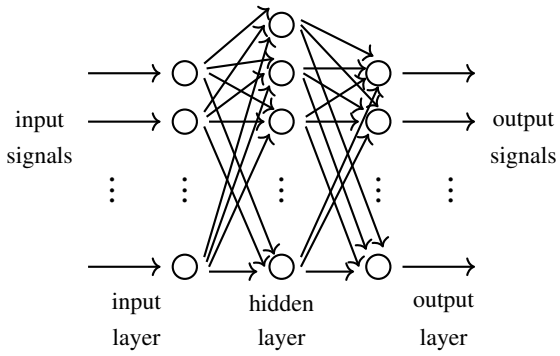

Fig. III.1. A simple neural network consists of exactly one hidden layer.

IV. Propagation and backpropagation

The problem of vanishing gradient is worse for recurrent neural networks. This is because each time step is the equivalent of an entire layer in a feedforward network. So training a recurrent neural network for $10,000$ time steps is equivalent to training a $10,000$-layer feedforward net, which leads to exponentially small gradients and a decay of information through time.

One way to address this problem is by the method of gating, which is a technique to help decide when to forget the current input, and when to remember it for future time steps. The most popular gating types are gated recurrent unit (GRU) and long short-term memory (LSTM), while other techniques include gradient clipping, steeper gates, and better optimizers.

When it comes to training a recurrent network, graphics processing units (GPUs) are preferred over central processing units (CPUs), via accumulated evidence on text processing tasks like sentiment analysis and helpfulness extraction. GPUs train the neural network 250 times faster, where GPUs can complete a computation in 1 day versus over 8 months using CPUs.

A notion called stacking (see Figure IV.1) consists of a more complex output than processing a single recurrent neural network.

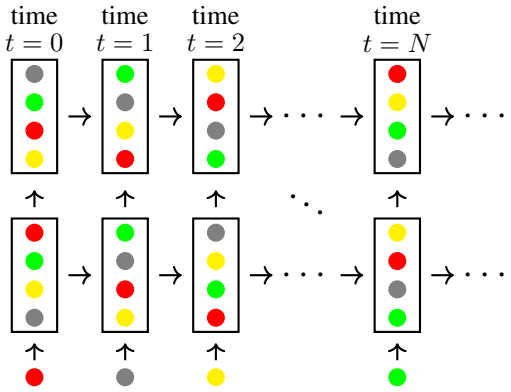

Fig. IV.1. An example of stacking in recurrent neural network.

V. The structure of neural network models

A network is a graph that represents a complex system, which uses linear and nonlinear algebraic operations, statistics, linear algebra, and any other necessary mathematical techniques to obtain accurate outcomes.

Assign a weight w_i to each edge (connection) between neuron from the first layer to our neuron, where $w_i \in \mathbb{R}$ is allowed to be any positive or negative number. Take all those activations a_i from the input (first) layer and compute their weighted sum:

$$\sum_{i=1}^{n} w_i a_i = w_1 a_1 + w_2 a_2 + \ldots + w_n a_n. \quad (1)$$

See Figure V.1.

Since the weighted sum $\sum_{i=1}^{n} w_i a_i$ is allowed to be any value, activation values should be between 0 and 1 in order to make sense, *e.g.*, for an image processing scenario. So in such example, normalize

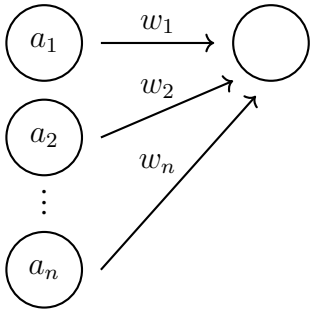

Fig. V.1. An instance of the input layer activations all connected to a node in the second layer.

the weighted sum into the range $(0, 1)$ using the logistic curve

$$\sigma : \mathbb{R} \to (0, 1), \text{ where } \sigma(x) = \frac{1}{1 + e^{-x}}.$$

The activation $\sigma(\sum_{i=1}^{n} w_i a_i)$ of the neuron is essentially a measure of the positivity of the relevant weighted sum. If the activation is meaningful only when the weighted sum is bigger than a bias b, then modify the activation by subtracting b: $\sigma(\sum_{i=1}^{n} w_i a_i - b)$. So the weights give what pixel pattern this neuron in the second layer is picking up on and the bias says how high the weighted sum needs to be before the neuron begins to be meaningfully active.

The above analysis is for a single neuron, so every neuron in the second layer is connected to all input neurons from the first layer, and each of those connections has its own weight (on the edge) and bias associated with it. This is just the connections from the first layer to the second layer. Connections between all the other layers have their own weights and biases associated with them.

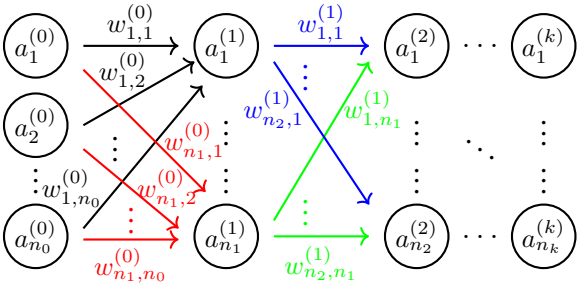

Fig. V.2. General neural network.

Representing this mathematically, consider the more general case in Figure V.2. Here, we have

$$\begin{aligned} a_1^{(1)} &= \sigma\left(w_{1,1}^{(0)}a_1^{(0)} + \ldots + w_{1,n_0}^{(0)}a_{n_0}^{(0)} + b_1^{(0)}\right) \\ a_2^{(1)} &= \sigma\left(w_{2,1}^{(0)}a_1^{(0)} + \ldots + w_{2,n_0}^{(0)}a_{n_0}^{(0)} + b_2^{(0)}\right) \\ &\vdots \qquad\qquad\qquad \vdots \\ a_{n_1}^{(1)} &= \sigma\left(w_{n_1,1}^{(0)}a_1^{(0)} + \ldots + w_{n_1,n_0}^{(0)}a_{n_0}^{(0)} + b_{n_1}^{(0)}\right) \\ &\vdots \qquad\qquad\qquad \vdots \end{aligned} \tag{2}$$

Rewrite (2) more compactly as matrices, *i.e.*,

$$\begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_{n_1}^{(1)} \end{pmatrix} = \widetilde{\sigma}\left(\begin{pmatrix} w_{1,1}^{(0)} & \ldots & w_{1,n_0}^{(0)} \\ w_{2,1}^{(0)} & \ldots & w_{2,n_0}^{(0)} \\ & & \\ w_{n_1,1}^{(0)} & \ldots & w_{n_1,n_0}^{(0)} \end{pmatrix}\begin{pmatrix} a_1^{(0)} \\ a_2^{(0)} \\ \vdots \\ a_{n_0}^{(0)} \end{pmatrix} + \begin{pmatrix} b_1^{(0)} \\ b_2^{(0)} \\ \vdots \\ b_{n_1}^{(0)} \end{pmatrix}\right),$$

or $a^{(1)} = \widetilde{\sigma}(W^{(0)}a^{(0)} + b^{(0)})$, where

$$\widetilde{\sigma}\left(\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}\right) := \begin{pmatrix} \sigma(v_1) \\ \sigma(v_2) \\ \vdots \\ \sigma(v_n) \end{pmatrix},$$

$$a^{(i)} = \begin{pmatrix} a_1^{(i)} \\ a_2^{(i)} \\ \vdots \\ a_{n_i}^{(i)} \end{pmatrix}, \quad \text{and} \quad b^{(i)} = \begin{pmatrix} b_1^{(i)} \\ b_2^{(i)} \\ \vdots \\ b_{n_{i+1}}^{(i)} \end{pmatrix},$$

etc. Figure V.2 has wt + biases parameters, where

$$\mathsf{wt} = a_{n_0}^{(0)}a_{n_1}^{(1)} + a_{n_1}^{(1)}a_{n_2}^{(2)} + \ldots + a_{n_{k-1}}^{(k-1)}a_{n_k}^{(k)}$$

$$\text{and} \quad \mathsf{biases} = \sum_{i=1}^{k} a_{n_i}^{(i)},$$

where wt is the number of weights and biases is the number of biases.

So for machine learning, we modify the computer with a valid setting for all of these weights and biases so that it will solve the original problem. It seems plausible to change the structure or improve the neural network in order to make the model more efficient. But if the network does work, and not for the reasons that we may expect, then investigating what the weights and biases are doing is an adequate way to challenge one's assumptions and explore the full spectrum of possible solutions.

As we move from the first layer of edges to the second layer of edges in the general neural network, see Figure V.2, we are applying matrix and column vector products and a logistic curve again, analogous to (2). A specific number that neurons hold depends on the input that has been fed into the model. So each neuron should be thought of as a function.

VI. Backpropagation Algorithm

Consider the general multilayered neural network in Figure V.2. Let $C_k = \sum_{j=1}^{n_k}(a_j^{(k)} - y_j^{(k)})^2$ be the cost function in the output layer, where $y_i^{(k)}$'s are the desired output.

Let $z_j^{(k)} = w_{j,1}^{(k-1)}a_1^{(k-1)} + \ldots + w_{j,n_{k-1}}^{(k-1)}a_{n_{k-1}}^{(k-1)} + b_j^{(k-1)}$, where $1 \le j \le n_k$. Let $a_j^{(k)} = \sigma_k(z_j^{(k)})$, where σ_k is a nonlinear function. The derivative of the cost function with respect to a weight is

$$\begin{aligned} \frac{\partial C_k}{\partial w_{j,u}^{(k-1)}} &= \frac{\partial z_j^{(k)}}{\partial w_{j,u}^{(k-1)}}\frac{\partial a_j^{(k)}}{\partial z_j^{(k)}}\frac{\partial C_k}{\partial a_j^{(k)}} \\ &= a_u^{(k-1)}\sigma_k'(z_j^{(k)})\left(2\left(a_j^{(k)} - y_j^{(k)}\right)\right), \end{aligned} \tag{3}$$

where $1 \le j \le n_k$ and $1 \le u \le n_{k-1}$.

The derivative of the cost with respect to one of the activations in layer $k-1$ is

$$\frac{\partial C_k}{\partial a_u^{(k-1)}} = \sum_{j=1}^{n_k}\frac{\partial z_j^{(k)}}{\partial a_u^{(k-1)}}\frac{\partial a_j^{(k)}}{\partial z_j^{(k)}}\frac{\partial C_k}{\partial a_j^{(k)}}, \tag{4}$$

where $1 \le u \le n_{k-1}$. Here, a neuron in layer $k-1$ influences the cost function through multiple paths, so we sum over activations in layer k. This tells us how sensitive the cost function is relative to the activation of the previous layer (see Figure VI.1).

Finally, we have

$$\frac{\partial C_k}{\partial b_j^{(k)}} = \frac{\partial z_j^{(k)}}{\partial b_j^{(k)}}\frac{\partial a_j^{(k)}}{\partial z_j^{(k)}}\frac{\partial C_k}{\partial a_j^{(k)}} = \sigma_k'(z_j^{(k)})2(a_j^{(k)} - y_j^{(k)}). \tag{5}$$

More generally,

$$\begin{aligned} \frac{\partial C_l}{\partial w_{j,u}^{(l-1)}} &= \frac{\partial z_j^{(l)}}{\partial w_{j,u}^{(l-1)}}\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}}\frac{\partial C_l}{\partial a_j^{(l)}} \\ &= a_u^{(l-1)}\sigma_l'(z_j^{(l)})2(a_j^{(l)} - y_j^{(l)}), \end{aligned} \tag{6}$$

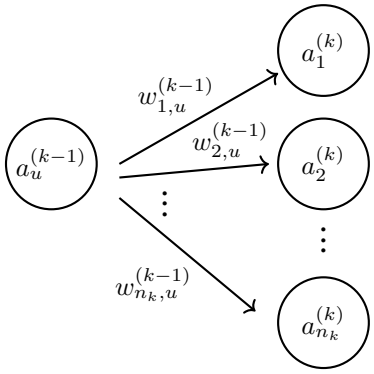

Fig. VI.1. The activation in the previous layer influences the cost function through multiple paths.

where $1 \leq l \leq k$.

Thus, the derivative of the full cost function

$$\frac{\partial C}{\partial w^{(u)}} = \frac{1}{N} \sum_{v=1}^{N} \frac{\partial C^{(v)}}{\partial w^{(u)}}$$

is the average of all training examples, and

$$\nabla C = \begin{pmatrix} \frac{\partial C}{\partial w_{1,1}^{(0)}} \\ \vdots \\ \frac{\partial C}{\partial b_{n_1}^{(0)}} \\ \vdots \\ \frac{\partial C}{\partial w_{1,1}^{(k)}} \\ \vdots \\ \frac{\partial C}{\partial b_{n_k}^{(k)}} \end{pmatrix}.$$

VII. Optimization and neural network reduction

Optimization on tactical computing platforms are very important in today's society ([3], [4], [9]–[11]), as more advanced models become quickly developed but computational resources remain limited. This prevents even the most advanced computing platforms to complete many of their algorithms.

Optimizers define how neural networks learn; they find the values of parameters/predictors such that a loss function is minimized. In general, the optimizers do not know the terrain of the loss so they need to minimize the function essentially blindfolded.

The original optimizer for deep neural networks involves taking small steps iteratively until the correct weights are reached. However, the problem is the weights are updated once after seeing the entire data set. So this gradient is typically large, and the weights could lead to larger jumps. It may also hover over its optimal value without actually being able to reach it. The solution to this is to update the parameters more frequently.

One example is to use stochastic gradient descent, which updates the weights after seeing each data point, instead of the entire data set. But this may make exceedingly noisy jumps that move away from the optimal values since it is influenced by *every* single sample. Because of this, mini-batch gradient descent is used as a compromise, updating the parameters after *several* samples.

Another way to decrease the noise of stochastic gradient descent is to add the concept of momentum. The parameters of a model may have the tendency to change in one direction, typically if examples follow a similar pattern. With this momentum, the model can learn faster by paying little attention to the few examples that throw it off from time-to-time. But there is a problem here also. The jumps could be too large to the point that the cost function actually moves away from the minimum, optimal values. That is, choosing to blindly ignore samples simply because they are not typical may be a costly mistake, resulting in a loss. However, adding an acceleration term will help. The weights for the model-in-training (neural network edge values that are being adjusted) could become larger, with little to no effect from the outliers. This would show that discarding the outliers would not lead to a drastic loss in order to fine-tune the current model.

With multiple predictors, the learning rate is fixed for every parameter, but one can impose adaptive learning rate to each parameter, where different size step is taken for every parameter. Adaptive learning rate optimizers are able to learn more along one direction than another, so they can traverse certain types of terrain. Furthermore, momentum updates for each parameter could also be imposed.

The mathematics behind neural networks reduces to using calculus and other analytical techniques to find the minimum of the cost function. Conceptually, we are thinking of each neuron as being connected to all the neurons in the previous layer. The weights w_i or $w_{i,j}^{(k)}$ in the weighted sum defining its activation in (1) and (2) are the strengths of those connections. The bias b is an indication of whether or not that

neuron tends to be active or inactive.

One constructs a cost function to determine wrong or erroneous output, i.e., it is the sum of the square of the differences between actual and predicted output activations. That is, a cost function C has $\mathsf{wt} + \mathsf{biases}$ as it input and a single number as its output, which says how bad its weights and biases are, and the way that it is defined depends on the network's behavior over all the thousands of pieces of labelled training data, which is the cost of a single training example.

We want to minimize the cost function by changing all of the weights and biases (all connections) to most efficiently decrease the cost. But the smallest local value of the cost function is not necessarily the global minimum value. In fact, attaining the global minimum may be computationally heavy, possibly wasting time and resources, since we are working with possibly millions of weights and biases. One uses gradient descent $-\nabla C(w_{1,1}^{(0)}, \ldots, w_{n_1,n_0}^{(0)}, \ldots, w_{1,1}^{(k)}, \ldots, w_{n_1,n_0}^{(k)}, b_1^{(k)}, \ldots, b_{n_{k-1}}^{(k)})$, which is the direction of the steepest descent, *i.e.*, it maximally decreases the output of the cost function as quickly as possible. For example, see Figure VII.1.

$$-\nabla C \begin{pmatrix} \begin{pmatrix} w_{1,1}^{(0)} \\ \vdots \\ w_{n_1,n_0}^{(0)} \\ \vdots \\ w_{1,1}^{(k)} \\ \vdots \\ w_{n_1,n_0}^{(k)} \end{pmatrix}^t \end{pmatrix} = \begin{pmatrix} 0.25 & \uparrow w_{1,1}^{(0)} \\ \vdots & \vdots \\ -0.05 & \downarrow w_{n_1,n_0}^{(0)} \text{ by } \approx 0 \\ \vdots & \vdots \\ 1.034 & \uparrow w_{1,1}^{(k)} \text{ by a lot} \\ \vdots & \vdots \\ -2.98 & \downarrow w_{n_1,n_0}^{(k)} \text{ by a lot} \end{pmatrix}$$

Fig. VII.1. The notation t is the transpose of the column vector. This is an example of the negative of the gradient function, paving a way for a strategy on a neural network to minimize the cost function. The relative magnitude of the components tells us how sensitive the cost function is to each weight and bias, *i.e.*, which changes matter more.

We see in Figure VII.1 that an adjustment to some of the weights have a greater impact on the cost function than the adjustment of other weights. Thus, some of the connections matter much more for the training data. The gradient descent function encodes the relative importance of each weight and bias. When changing by some small multiple of $-\nabla C$, the relative importance of the weights and bias remain the same. As we can see, this technique is computationally heavy due to immense set of input data. Furthermore, the testing data may show that the number of correct outputs over the total number of simulations may be close to approximately $95 - 98\%$ of testing accuracy, not 100%, as we would like.

Backpropagation is an algorithm for computing the gradient descent. Theoretically, the way we adjust weights and biases for a single gradient descent step also depends on every single training example (since each step uses every example), but for computational efficiency, we will keep from needing to obtain every single example for every single step.

First, consider one single example, for example, the image of the number 8. We will discuss the effects this one training example have on how the weights and biases get adjusted. If the neural network is not yet well-trained, activations in the output will appear random. For example, there are 10 outputs labelled from 0 to 9 and we may have

$$\begin{aligned} a_1^{(k)} &= 0.4 \\ a_2^{(k)} &= 0.7 \\ a_3^{(k)} &= 0.2 \\ a_4^{(k)} &= 0.1 \\ a_5^{(k)} &= 0.0 \\ a_6^{(k)} &= 0.4 \\ a_7^{(k)} &= 1.0 \\ a_8^{(k)} &= 0.1 \\ a_9^{(k)} &= 0.0 \\ a_{10}^{(k)} &= 0.3 \end{aligned}$$

where $a_j^{(k)}$ is the neural network output for the number $j-1$, $1 \leq j \leq 10$. If the network is well-trained, then all $a_j^{(k)}$ should go down to 0, except $a_9^{(k)}$ since we are considering the image of the number 8, so $a_9^{(k)}$ should go up to 1. We cannot change these activations but instead, we may influence weights and biases. Moreover, the sizes of these nudges are proportional to how far away each current value is from its target value. For example, the increase to that number 8 neuron activation, *i.e.*, $a_9^{(k)}$, is more important than the decrease to the number 3 neuron, *i.e.*, $a_4^{(k)}$, which is fairly close to where it should be.

Focusing on $a_9^{(k)}$, this activation $a_9^{(k)}$ is defined as the weighted sum of all of the activations in the previous layer, plus a bias, and then it has been

substituted into a nonlinear function σ, *cf.*, (2). We see that there are three different ways to increase the activation: increase the bias, increase the weights, or change the activations from the previous layer.

Consider the weights. In order to adjust the weights, notice that the weights have differing levels of influence. The connections with the brightest neurons from the preceding layer have the biggest effect since those weights are multiplied by larger activation values. If we increase one of those weights, it has a stronger influence on the ultimate cost function than increasing the weights of connections with dimmer neurons.

Note that when we consider the gradient descent, we care not only the sign of each component, but we care about the magnitude of each component. That is, the neurons that are firing when seeing the number 8 get even more greatly linked to those firing when thinking about the number 8.

The last way to increase this neuron's activation is by changing all the activations from the previous layer, and in proportion to each associated weight. Namely, if everything connected to that digit 8 neuron with a positive weight got brighter and everything connected with a negative weight got dimmer, then the digit 8 neuron would become more active. We do not have direct influence on these activations, but we only have control over the weights and biases.

Now, focusing on all the other neurons $a_j^{(k)}$ in the output layer, where $j \neq 9$, we want all of the other neurons in the last layer to become less active; each of those other output neurons has its own algorithm about what should happen to that second-to-last layer. We thus need to put together the strategy for the output neuron 8 along with the strategies for all the other output neurons for what should happen to this second-to-last (hidden) layer, proportionately to the weights and to how much each of those neurons needs to change.

Applying backpropagation is adding together these desired effects to obtain a list of nudges that we want to occur in the second-to-last layer. After we apply this, we recursively apply this algorithm to the relevant weights and biases that determine those values by repeating this process and moving backwards through the network.

This is how a single training example wishes to nudge each one of those weights and biases. If we only focused on one simple example of 8, then the network would ultimately be incentivized just to classify all images as an 8. So what we need to do is go through this same backpropagation routine for every other training example, recording how each of them would like to change the weights and the biases. Then we average together these desired changes, over all training data. This collection of the average changes to each weight and bias is the negative gradient of the cost function.

Gradient descent in practice takes large amount of computational time to add up the influence of every single training example, every single gradient descent step. So what we do instead is stochastic gradient descent, *i.e.*, we randomly shuffle our training data and then divide it into groups of mini-batches, for example, each one having 1000 training examples. Then we compute a gradient descent step using backpropagation according to the mini-batch. This doesn't give the actual gradient descent of the cost function, which depends on all of the training data. So this is not the most efficient step downhill, but we do obtain a fairly decent approximation, and it gives us a significant computational speed up. That is, if we plot the trajectory of the network under the relevant cost surface, it would be similar to a person aimlessly walking downhill, taking quick steps, rather than a carefully calculating person determining the exact downhill direction of each infinitesimal step, before taking a very slow and careful step in that direction.

VIII. Neural network complexity

Numerous labelled training data are needed, like handwritten numbers, letters, and symbols, or people labelling tens of thousands of images. Neural networks are governed by bandwidth and performance. As there is a higher demand for artificial intelligence, there is a greater need for bandwidth reduction and performance bump, which will allow one to load less data from system memory into the local memory and overall from the system.

Although researchers have investigated many ways towards neural network reduction, we focus on certain convolution called Rectified Linear Units (ReLU) or rectified linear activation function. That is, we insert a new layer to create more sparsity in the weights and in the activations. ReLU is a piecewise linear function whose output will be the input directly

if it is positive, or else, its output is zero. That is, it's the linear function

$$f(x) = \begin{cases} 0 & \text{if } x \leq 0, \\ x & \text{if } x > 0. \end{cases}$$

However, for our manuscript, we modify the function as

$$f(x) = \begin{cases} 0 & \text{if } x \leq \varepsilon, \\ x & \text{if } x > \varepsilon, \end{cases} \tag{7}$$

where $\varepsilon > 0$. After applying modified ReLU, we know that sparsity increases from 15% to about 35% in the weights after pruning and then retraining, and in activation, applying modified ReLU (pruning) and then retraining will extend sparsity from 40% to approximately $60-90\%$. It's clear that our modification will increase the sparsity to beyond 35% and 90% since the weights and activations that contribute negligible amounts still contribute negligible amounts after pruning; this is also because the sum of negligible numbers is still negligible relative to other weights and activations. This procedure will give us a swift identification of objects by machines as complexity has been further reduced by using modified ReLU.

Therefore, we modify (7) and apply it to nonlinear functions like the sigmoid in (2) so that modified ReLU is applied to our deep neural network, i.e.,

$$\begin{aligned} a_1^{(1)} &= \begin{cases} 0 & \text{if } x_1^{(1)} \leq \varepsilon_1^{(1)}, \\ \sigma(x_1^{(1)}) & \text{if } x_1^{(1)} > \varepsilon_1^{(1)}, \end{cases} \\ a_2^{(1)} &= \begin{cases} 0 & \text{if } x_2^{(1)} \leq \varepsilon_2^{(1)}, \\ \sigma(x_2^{(1)}) & \text{if } x_2^{(1)} > \varepsilon_2^{(1)}, \end{cases} \\ &\vdots \qquad\qquad\qquad \vdots \\ a_{n_1}^{(1)} &= \begin{cases} 0 & \text{if } x_{n_1}^{(1)} \leq \varepsilon_{n_1}^{(1)}, \\ \sigma(x_{n_1}^{(1)}) & \text{if } x_{n_1}^{(1)} > \varepsilon_{n_1}^{(1)}, \end{cases} \\ &\vdots \qquad\qquad\qquad \vdots \end{aligned} \tag{8}$$

where $\varepsilon_1^{(1)}, \varepsilon_2^{(1)}, \ldots, \varepsilon_{n_1}^{(1)}, \ldots \geq 0$, and

$$\begin{aligned} x_1^{(1)} &= w_{1,1}^{(0)} a_1^{(0)} + \ldots + w_{1,n_0}^{(0)} a_{n_0}^{(0)} + b_1^{(0)}, \\ x_2^{(1)} &= w_{2,1}^{(0)} a_1^{(0)} + \ldots + w_{2,n_0}^{(0)} a_{n_0}^{(0)} + b_2^{(0)}, \\ &\vdots \qquad\qquad\qquad\qquad \vdots \\ x_{n_1}^{(1)} &= w_{n_1,1}^{(0)} a_1^{(0)} + \ldots + w_{n_1,n_0}^{(0)} a_{n_0}^{(0)} + b_{n_1}^{(0)}, \\ &\vdots \qquad\qquad\qquad\qquad \vdots \end{aligned}$$

IX. Summary and Future Direction

Deep neural networks are emerging in many technologies today. We introduced three common neural networks and then discussed the role of activations, weights, hidden layers, and bias. Understanding the mathematics behind propagation and backpropagation enabled us to introduce an optimization model which further reduces the computational complexity by introducing more sparsity into the model. With modified ReLU implemented in the deep neural network, we have enabled for the training and inference speeds to accelerate further on computing platforms consuming low computational resources.

References

[1] H. Abdi. A neural network primer. *Journal of Biological Systems*, 2(03):247–281, 1994.

[2] M. Anthony and P. L. Bartlett. *Neural network learning: Theoretical foundations*. cambridge university press, 2009.

[3] V. R. Dasari, M. S. Im, and L. Beshaj. Solving machine learning optimization problems using quantum computers. *arXiv preprint arXiv:1911.08587, to appear in Proc. SPIE*, pages 1–6, 2019.

[4] V. R. Dasari, M. S. Im, and B. Geerhart. Complexity and mission computability of adaptive computing systems. *The Journal of Defense Modeling and Simulation*, 17(1):1–7, 2019.

[5] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376, 2006.

[6] A. Graves and J. Schmidhuber. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural networks*, 18(5-6):602–610, 2005.

[7] L. K. Hansen and P. Salamon. Neural network ensembles. *IEEE transactions on pattern analysis and machine intelligence*, 12(10):993–1001, 1990.

[8] R. Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[9] M. S. Im and V. Dasari. Genetic optimization algorithms applied toward mission computability models. *88th MORS Symposium: AI and Autonomous Systems*, WG35:1–11, 2020.

[10] M. S. Im and V. R. Dasari. Optimization and synchronization of programmable quantum communication channels. In E. Donkor and M. Hayduk, editors, *Quantum Information Science, Sensing, and Computation X*, volume 10660, pages 166–172. International Society for Optics and Photonics, SPIE, 2018.

[11] M. S. Im, V. R. Dasari, L. Beshaj, and D. Shires. Optimization problems with low SWAP tactical computing. In M. Blowers, R. D. Hall, and V. R. Dasari, editors, *Disruptive Technologies in Information Sciences II*, volume 11013, pages 70–76. International Society for Optics and Photonics, SPIE, 2019.

[12] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. In *Proceedings of the 31st international conference on neural information processing systems*, pages 972–981, 2017.

[13] Z. C. Lipton, J. Berkowitz, and C. Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, pages 1–38, 2015.

[14] H. A. Rowley, S. Baluja, and T. Kanade. Neural network-based face detection. *IEEE Transactions on pattern analysis and machine intelligence*, 20(1):23–38, 1998.