DEEP NEURAL NETWORKS AND UNIVERSAL APPROXIMATORS II

Ying Liu, Savannah State University; Majid Bagheri, Savannah State University;
Antonio Velazquez, Savannah State University; Asad Yousuf, Savannah State University

Abstract

There are many studies of approximations using deep neural networks. In this paper, the authors provide yet another proof that deep neural networks are universal approximators. In their earlier work, the authors showed that an arbitrary binary target function can be effectively rewritten in terms of a set of strings, or a set of subsets, and that a single hidden neuron can identify and only identify a single string or a single subset. Therefore, an arbitrary binary target function can be effectively rewritten in the form of a neural network with one hidden layer. In this study, the authors imposed locality on the deep neural network, and will show here that an arbitrary binary target function can be effectively rewritten in the form of a locally connected deep neural network that can have many hidden layers. Although it will increase the neural network size, as a neural network is localized, it will generally increase the speed of training for large networks.

Key words: AI, Universal Approximator, Completeness, Deep Neural Network, Machine Learning, Supervised Learning, Unsupervised Learning, Locally Connected.

Introduction

Neural networks provide good solutions to many supervised learning problems. Neural networks have a long history, but there have been two main developments in recent years, deep learning and transforming (Amari, Kurata & Nagaoka, 1992; Byrne, 1992; Kubat, 2015). In 2006, authors in several other studies introduced the idea of "deep" neural networks (Hinton, Osindero & Teh, 2006; LeCun, Bengio & Hinton, 2015; Bengio, 2009; Coursera, 2017; Bengio, Courville & Vincent, 2013; Schmidhuber, 2015; Ciresan, Meier & Schmidhuber, 2012). Examples of software include TensorFlow (TensorFlow, 2017), Torch (Torch 2017), and Theano (Theano, 2017). Layers in deep neural networks (DNNs) serve as the building blocks of the architecture, enabling the model to learn from the data. Each layer has a specific function in transforming the input into an output, progressively extracting higher-level features. For example, early layers might detect edges or simple patterns (e.g., in images), while middle layers may capture more complex patterns (e.g., shapes or textures), and deeper layers identify task-specific, high-level features (e.g., faces or objects). The transformer model, introduced by Vaswani et al. (2017), represents a significant advancement in deep learning architectures, particularly in natural language processing. Each layer in the DNN is replaced by a transformer (Vaswani et al., 2017).

The transformer leverages a fully self-attentive mechanism to model complex dependencies between elements of a sequence. This architecture enables the transformer to be trained more efficiently and with greater parallelism, leading to faster training times and improved scalability. As a result, the transformer has become the backbone of numerous state-of-the-art models, including Chat GPT (OpenAI, 2023) and Claude (Anthropic, 2023), profoundly influencing the development of modern deep learning systems. Studies of the neural network as universal approximators have a long history. Hornik, Stinchcombe, and White (1989) established models showing that multi-layer feed-forward networks with hidden layers using arbitrary squashing functions are capable of approximating any measurable function from one finite dimensional space to another to any desired degree of accuracy, provided that a sufficient number of hidden units are available. In this sense, multi-layer feed-forward networks are a class of universal approximators.

Hinton, Osindero, and Teh (2006) introduced the idea that deep belief networks (DBN) are generative neural network models with many layers of hidden explanatory factors, along with a greedy layer-wise unsupervised learning algorithm. The building block of a DBN is a probabilistic model called a restricted Boltzmann machine (RBM), which is used to represent one layer of the model. Restricted Boltzmann machines are interesting, because they have been successfully used as building blocks for training deeper models. Le Roux and Bengio (2008) proved that adding hidden units yields a strictly improved modeling power, and that RBMs are universal approximators of discrete distributions.

Liu and Wang (Liu, 1993; Liu, 1995; Liu, 1997; Liu, 2002; Liu & Wang, 2018; Liu, 2018a/b) proved that DNNs implement an expansion and the expansion is complete; a complete expansion can be used to expand any target functions. Cheng, Li, and Lu (2019) introduced a type of convolutional neural network (CNN) that can implement the Fourier and local Fourier transformations for approximation in a large class of problems. Cybenko (1989) showed that a finite sum of any continuous sigmoid function can be used to approximate any univariate function using functional analysis. Liu and Yousuf (2020) showed that DNNs are effective universal approximators. An arbitrary binary target function can be effectively rewritten in the form of a DNN; thus, proving that DNNs can implement any target mappings. An example of a locally connected network is the convolutional neural network (CNN) (LeCun, Bottou, Bengio & Haffner, 1998; Krizhevsky, Sutskever & Hinton, 2012), which is a specialized type of deep learning model particularly well-suited for processing images.

It utilizes convolutional layers that apply filters across the input data to capture spatial hierarchies of patterns. This architecture allows the network to automatically learn and detect features such as edges, textures, and objects within images, making them highly effective for tasks such as image classification, object detection, and segmentation. Several approaches can be applied for reducing computation times of neural networks, including model optimization, hardware utilization, and algorithmic refinement. 1) Model optimization techniques, such as pruning, quantization, and knowledge distillation, reduce model size while maintaining performance (Han, Pool, Tran & Dally, 2015). Weight sharing and sparse representations are also effective in minimizing redundancy in parameters. 2) Efficient architectures, for example MobileNet (Howard et al., 2017) and EfficientNet (Tan & Le, 2019) were explicitly designed to reduce computational overhead through depth-wise separable convolutions and scaling strategies. 3) Hardware-specific optimizations are accelerators—such as GPUs, TPUs, and custom ASICs—that exploit parallelism and optimized memory access patterns to enhance speed (Jouppi et al., 2017). 4) Training techniques, such as mixed precision training (Micikevicius et al., 2018), reduce floating-point precision for faster computation, while learning rate schedulers and gradient accumulation ensure efficient convergence.

In this paper, the authors will show that an arbitrary binary target function can be effectively rewritten in the form of a locally connected DNN. The result opens a discussion for exploring an approach of locally connected neural networks as an alternative to globally connected models. Additionally, the authors will build on their earlier work (Liu & Yousuf, 2020); in particular, locality, will be imposed on the neural network. As a comparison, the earlier work had a single globally connected network with one hidden layer, while the work presented here represents many hidden layers with locally connected neural networks. The author will show that an arbitrary binary target function can be effectively rewritten in the form of a locally connected DNN. To prove this result: 1) the earlier work by the authors will be briefly reviewed foundational to this current study; 2) the result is proof for a special case—a binary locally connected network; and, 3) the result will be proven by removing the binary condition. For a given target function, there are many effective ways to construct a locally connected DNN.

The results, then, open a discussion for exploring an approach of locally connected neural networks as an alternative to globally connected models. The von Neumann bottleneck refers to the limitation in computing systems that stems from the separation of the central processing unit (CPU) and memory in the von Neumann architecture. Increasingly, both computation times and electric powers are spent on moving data from one place to another. For example, electric power consumption has been increasing rapidly for the transformer models. (Wall Street Journal, 2024). One of the main reasons that transformer models use far more power than biological neurons is that the biological systems are locally connected networks. In this paper, the authors show that, as the models transit from globally connected networks to locally connected networks, the computing power will not be decreased, but the amount of data transfer can be reduced.

Review: Effectively Rewriting a Mapping with One Hidden Layer

An arbitrary binary target function can be effectively rewritten in terms of a set of strings, or a set of subsets. A single string or a single subset can be identified by a single hidden neuron, and this neuron will only identify the string or the subset; therefore, an arbitrary binary target function can be effectively rewritten in the form of a neural network with one hidden layer (Liu, Yousuf, 2020). A binary-neuron input instance is 00 … 0, or, 0 … 01, … (Amari et al., 1992; Byrne, 1992; Kubat, 2015) and an instance space (Kubat, 2015) is a set of all instances given by Equation 1:

$$X = \{0\ldots00, 0\ldots01, 0\ldots10, 0\ldots11, \ldots\} \tag{1}$$

Given an arbitrary binary target function, it can be effectively rewritten in terms of a set of strings, or a set of subsets given by Equations 2 and 3:

$$h = \{s_0, s_1, s_2, \ldots\} \tag{2}$$

$$s_i \subseteq \{0, 1, 2, \ldots d-1\} \tag{3}$$

Example. Given a function in Table 1, the mapping can be rewritten using Equations 4 and 5:

Table 1. A sample binary function with three inputs.

x_0	x_1	x_2	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

$$y = \{001, 011, 100\} \tag{4}$$

$$y = \{\{2\}, \{1, 2\}, \{0\}\} \tag{5}$$

where, y is overloaded with a table, a mapping, a set of strings, and a set of subsets, and s_i in Equation 3 is overloaded with a string and a subset.

Without loss of generality, it can be assumed that there is only one output variable for now. For the case of multiple output variables, it can be treated as multiple mappings. The neural network used in this review section will have one input layer, one output layer, and one hidden layer. The neuron values are given by Equation 6 (Amari, Kurata & Nagaoka, 1992; Byrne, 1992; Kubat, 2015):

$$y_i = f\left(\sum w_{ij} x_j + b_i\right), i = 1, 2, 3, \ldots \tag{6}$$

where, f is a sigmoid function given define by Equation 7:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{7}$$

To compute the connection weights, a constant L is introduced; without a loss of generality, set L = 10. For an arbitrary target function, it can be rewritten in the form of Equations 2 and 3. The rules for construction of a DNN are:

1. The DNN will have one input layer, one output layer, and one hidden layer. The input layer has d neurons.
2. Each neuron in the hidden layer identifies one string in a target function, h = {s_0, s_1, …, }, so the number of neurons in the hidden layer is |h|, which is the number of strings or the number of subsets.
3. The output layer has one neuron; the neuron value is 1, if any one of the hidden layer neurons is 1.
4. Assume that s is a subset in a mapping, h; and assume a hidden neuron will identify s; the subset, then, is given by Equations 8 and 9:

$$s = \{j_0, j_1, j_2, \ldots\} \tag{8}$$

$$s \subseteq \{0, 1, 2, \ldots d-1\} \tag{9}$$

The hidden neuron has weights and biases as follows:

set weight = L, for input neurons {j_0, j_1, j_2, …}
set weight = – L, for all other input neurons
set bias = - (|s| - 1) · L

It has been proven that this simple ANN will implement a target function (Liu & Yousuf, 2020). To summarize:

1. An arbitrary binary target function can be effectively rewritten in terms of a set of strings, or a set of subsets, given by Equations 2 and 3:
2. A single hidden neuron can identify and only identify a single string or a single subset. The weights and biases are directly determined from a given target function by the rules in this section.
3. An arbitrary binary target function can be effectively rewritten in the form of a neural network with one hidden layer.

Binary Locality or Bilocality

In a locally connected neural net, let the maximum number of connections of a hidden neuron be N. To simplify the discussion, let the locality be extreme: N = 2; from Figure 1, this is called binary locality or simply bilocality. Once N is restricted, the size of connection matrices is restricted, at the cost of increasing the number of matrices. This reduction of one large matrix into many smaller matrices has its implications in computation efficiency, especially when the matrix is very large.

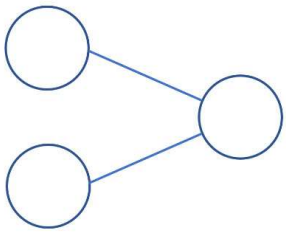

Figure 1. A hidden neuron has two input neurons.

The following naming convention will be adopted:

- The hidden layer closest to the input is the first hidden layer.
- The hidden layer closest to the output is the last hidden layer.

Assumption 1:
The DNN (deep neural network) is bilocally connected, where each hidden neuron can have only two or fewer connections.

Assumption 2:
The number of neurons in the input layer is a power of 2 (e.g., 2, 4, 8, 16, …).

These two assumptions will be removed later. Furthermore, Assumption 1 only applies to the hidden neurons; the connections of the single output neuron are determined by the number of strings in a target function in Equation 2, |h|. Without loss of generality, it can be assumed that there is only one output variable for now. For the case of multiple output variables, it can be treated as multiple mappings. In a binary locally connected network (bilocal network), it is only natural to group the connection weights of a neuron with the neuron rather than group them into connection matrices. A binary locally connected neural net is a set of neurons; a neuron has a neuron value, two connection weights, and a bias (called neuron-based computation): NN1 = {Neuron}, Neuron = {value, w0, w1, b}. This is in contrast to the view that a neural net is a set of neurons (where each neuron has a single value), a set of connection matrices, and a set of bias vectors (called matrix-based computation): NN2 = {Neurons, Matrix, Bias}. To compute the connection weights, a constant L will be introduced; without a loss of generality, set L = 10. A binary function is then rewritten in terms of a set of strings of 0's and 1's. A string in the set is directly imposed to the input neurons. For a bilocal network, two input neurons are grouped together and its two-bit pattern is passed to a hidden neuron in the first hidden layer.

Let a sample string be $x_0x_1x_2x_3$, where the pattern x_0x_1 can be identified by a hidden neuron, h_1, in the next layer, and the pattern x_2x_3 can be identified by a hidden neuron, h_2. The identification of x_0x_1 is propagated to the next layer via h_1, and the identification of x_2x_3 to h_2. To identify the entire pattern $x_0x_1x_2x_3$, h_1 and h_2 are further propagated to a hidden neuron in the next layer, say h_3, which only needs to identify the pattern "11" (i.e., both h_1 and h_2 have identified their required patterns). This is the basic idea of the newly proposed algorithm. The new rules for the network construction are:

1. The input layer has $d = 2^K$ neurons. The DNN has one input layer, one output layer, and O (log d) hidden layers.
2. Each neuron in the last hidden layer identifies one string in a target function, $h = \{s_0, s_1, \ldots, \}$, so the number of neurons in the last hidden layer is $|h|$, which is the number of strings in Equation 2 or the number of subsets.
3. The output layer has one neuron; the neuron value is 1, if any one of the last hidden layer neurons is 1.

Rule 1 states that there are *d* input neurons. The condition, $d = 2^K$, is for the sake of easy discussion and will be removed later. Rule 2, together with several other rules, describes the overall hidden neuron structures; each layer has a specific function in transforming the input into an output, progressively identifying bigger bit patterns for strings in a target function. In particular, Rule 2 specifies the last hidden layer, and its role is: a) the number of hidden neurons in the last hidden layer is the same as the number of strings in a given target function, and b) each hidden neuron in the last hidden layer will identify and only identify one string in the target function. Rule 3 describes the output layer. For the sake of this discussion, assume that there is only one output variable, per our earlier assumption, so there is only one output neuron. If an input string is one of the strings in a target function, one of the hidden neuron values in the last hidden layer is 1, which will cause the output neuron to be 1. If an input string is not in the target function, all of the hidden neurons in the last hidden layer will be 0, which will cause the output neuron to be 0.

Single String Identification

To identify a single string or a single subset, let the input layer have *d* neurons; let the first hidden layer have d/2 hidden neurons; let the second hidden layer have d/4 hidden neurons; and, let the last hidden layer have one hidden neuron. The input layer and all hidden layers together then form a binary tree, called a hidden tree. A hidden tree will identify one string in a target function later. In a complete binary tree, there exist relationships between the height, the number of edges, and the number of nodes in each layer from which a complete binary tree has:

- *d* input neurons
- log(d) hidden layers
- 2d – 2 weights
- d – 1 hidden neurons

By way of example, Figure 2 shows a hidden tree that has:

- d = 4 input neurons
- log(d) = 2 hidden layers
- 2d – 2 = 6 weights
- d – 1 = 3 hidden neurons

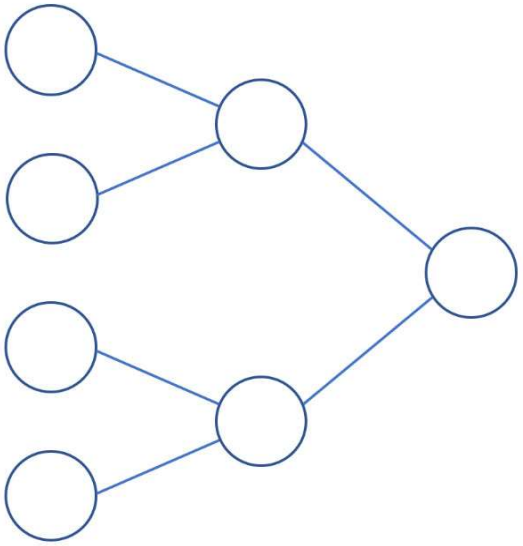

Figure 2. An example of a hidden tree with 4 input neurons.

In a complete binary tree, there exist relationships between the height, the number of edges, and the number of nodes in each layer. The four input neurons are drawn in column 1; thus, d = 4. Shown in Figure 2, the number of hidden layers is log (d) = 2: column 2 and column 3. Also shown in Figure 2 as edges is the number of weights: 2d – 2 = 6. The number of hidden neurons is d – 1 = 3: the 3 nodes in columns 2 and 3. This is the tradeoff between a globally connected network and a locally connected network. There are two costs of locality: 1) from using a single hidden neuron to identify a single string in a fully connected network to d – 1 neurons; 2) from using *d* weights to identify a single string in a fully connected network to 2d – 2 weights.

The first hidden layer identifies the input patterns. Each neuron in the first hidden layer identifies two input bits (bilocal). The number of neurons in the first hidden layer has d/2 neurons. After all of the 2-bit patterns are identified, the results propagate up, eventually to one single neuron in the last hidden layer. The role of the first hidden layer is to identify a single string or a single subset, and the roles of the rest of the hidden layers are to pass the results of the first hidden layer to a single root of a hidden tree in the last hidden layer.

Target Function Identification

Each neuron in the last hidden layer identifies one string in a target function, $h = \{s_0, s_1, \ldots, \}$, so the number of neurons in the last hidden layer is $|h|$, which is the number of strings or

the number of subsets. Figure 3 shows how each of the neurons in the last hidden layer grows a binary tree all the way to the input neurons.

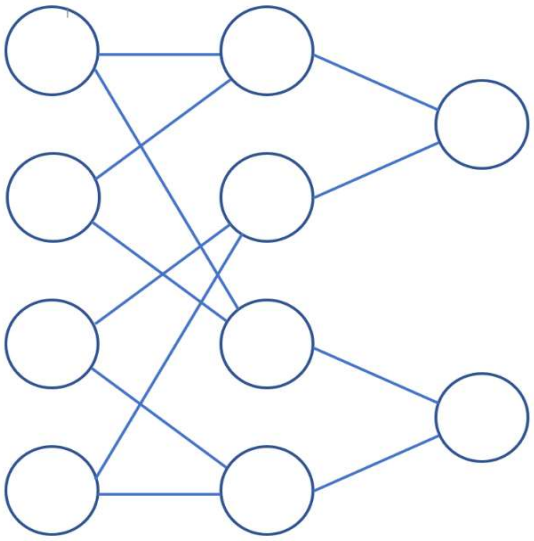

Figure 3. An example of two hidden trees for two strings.

In this example, the input layer has d = 4 input neurons and the target function has two strings to be recognized. There are two neurons in the last hidden layer; each is responsible for identifying one string. Each of the two neurons in the last hidden layer forms a binary tree. Within each tree, there are d – 1 hidden neurons in log(d) hidden layers and *d* input neurons. The rules for the hidden trees are:

4. Each of the neurons in the last hidden layer grows a binary tree all the way to the input layer neurons. There are log(d) hidden layers, where *d* is the number of input neurons. There are (d – 1) hidden neurons in each hidden binary tree.
5. The first hidden layer identifies the input patterns. Each neuron in the first hidden layer identifies two input bits (bilocal).

Let *s* be a single subset that is given in Equations 2 and 3, such that the rule for neurons in the first hidden layer is:

6. Assume that *s* is a subset in a mapping, *h*; further assume that a hidden neuron identifies *s*. In this case, the subset is given by Equations 8 and 9:

The hidden neuron has weights and biases as follows:

set weight = L, for input neurons $\{j_0, j_1, j_2, \ldots\}$
set weight = – L, for all other input neurons
set bias = $-(|s| - 1) \cdot L$

After all of the 2-bit patterns are identified in the first hidden layer, based on the rules above, the results will propagate up, eventually to one single neuron in the last hidden layer for one string/subset in Equations 2 and 3. The rule for neurons in the rest of the hidden layers is:

7. For the rest of the hidden layers (other than the first), all connection weights are L and all biases are $-(|s| - 1) \cdot L$, which is -L for bilocal hidden neurons.

This is an effective construction of a bilocal DNN from a given target function, which will be justified in the next section.

Effectively Rewriting a Mapping in Terms of a Bilocal Deep Neural Network

In the earlier review section, it was noted that a 2-bit pattern can be identified correctly by a hidden neuron. In this paper, the authors first identify a 2-bit pattern by one neuron in the first hidden layer, which has been proven to be correct. Second, the above step is repeated for all 2-bit patterns in the input layer. For d-input neurons, there are d/2 neurons in the first hidden layer. This step is already different from the authors' previous study in which they used one neuron instead of d/2 neurons. Third, the results of the first hidden layer simply propagate up. Let h_1 and h_2 be two neurons in the first hidden layer, the weights and the biases of a bilocal neuron, h_3, in the next hidden layer are simply (L, L), and -L, respectively, which identify the pattern "11" (i.e., both h_1 and h_2 have identified their required patterns). Each neuron in the second hidden layer identifies a 4-bit pattern. Fourth, each neuron in the third hidden layer identifies an 8-bit pattern, …, eventually, each neuron (root of a hidden tree) in the last hidden layer identifies one string or one subset. Finally, the output layer has one neuron; the neuron value is 1, if any one of the last hidden layer neurons is 1.

Since bilocal neurons are so simple, details can be worked out from the beginning in just a few lines. One bilocal hidden neuron is given in Equation 10:

$$m = f\left(a_0 x_0 + a_1 x_1 + b\right) \tag{10}$$

where, *f* is given in Equation 7.

Each hidden neuron in the first hidden layer has two weights and there are d/2 neurons. Since there are only four possible patterns to be identified, Table 2 lists the parameters in Equation 10 required for each case.

Table 2. Bilocal hidden neuron parameters for all 2-bit identifications.

Pattern to be identified	a_0	a_1	b
00	-L	-L	L
01	L	-L	0
10	-L	L	0
11	L	L	-L

Take, for example, one instance in detail. Assume that a neuron, *m*, can identify a pattern, "10"; from Table 2, Equation 10 is changed to Equation 11. All possible inputs and outputs for Equation 11 are listed in Table 3.

$$m = f\left(-Lx_0 + Lx_1\right) \tag{11}$$

Table 3. Inputs and outputs for Equation 11.

Input	$-Lx_0 + Lx_1$	m	int (m)
00	0	1/2	0
01	-L	0	0
10	L	1	1
11	0	1/2	0

Column 1 shows all possible inputs for 2-bit patterns. Column 2 is the intermediate step. Column 3 shows the neuron values. Column 4 takes the integer part of Column 3. In Table 3, int(m) is the integer function in C# language. The hidden neuron identifies the correct string, "10", by Equation 12:

$$m = f\left(-Lx_0 + Lx_1\right) = \frac{1}{1+e^{-L}} \approx 1 \tag{12}$$

If there is a single bit difference ("00", "11"), the hidden neuron has a value given by Equation 13:

$$m = \frac{1}{1+e^{0}} \approx 0.5 \tag{13}$$

If there is a 2-bit difference ("01"), the hidden neuron has a value given by Equation 14:

$$m = \frac{1}{1+e^{L}} \approx 0 \tag{14}$$

In general, if an input string differs from the string, *s*, by 0 bits, 1 bit, 2 bits, 3 bits, etc., the hidden neuron identifies the string with values given in Equation 15:

$$m = 1, 0.5, 0, 0, \ldots \tag{15}$$

This hidden neuron can clearly identify, and only identify, one string or one subset, *s*. Consider this next example. Let a given target function hold the four inputs given in Table 4.

Table 4. A sample binary function with four inputs.

x_0	x_1	x_2	x_3	y
0	0	1	1	1
1	0	0	1	1

The rest of the rows in Table 4 all have y(x) = 0. The strings are y = {0011, 1001}, and the set of subsets is y = {{2,3}, {0,3}}. Table 5 gives the weights and biases of the hidden neurons (a_0, a_1, b in Equation 10) in two hidden trees. Each row specifies all parameters in a hidden tree. Column 1 is the input string. Column 2 (m_0) and Column 3 (m_1) are hidden neurons in the first hidden layer. Column 4 (m_2) is the hidden neuron in the last hidden layer.

Here, m_0 and m_1 are hidden neurons in the first hidden layer and m_2 is the one in the last hidden layer. Figure 4 shows that there is one tree for each string/subset.

Table 5. The weights and biases of the hidden neurons for two strings.

String	m_0	m_1	m_2
0011	(-L,-L,L)	(L,L,-L)	(L,L,-L)
1001	(-L,L,0)	(L,-L,0)	(L,L,-L)

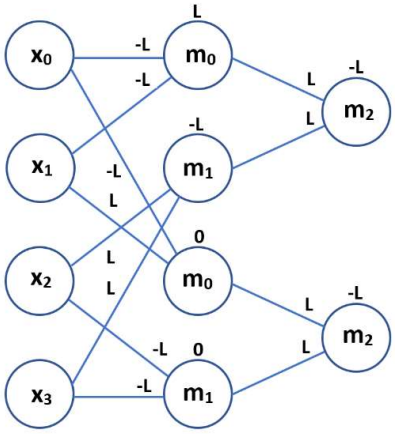

Figure 4. An example of two hidden trees for two strings: 0011 and 1001.

A target function is written in terms of a set of strings. For each string in the target function, there is one hidden tree that can identify it and only it. In this example, there are four inputs: x_0, x_1, x_2, and x_3 and the strings in a target function are y = {0011, 1001}. Here, m_0 and m_1 are hidden neurons in the first hidden layer and m_2 is in the last hidden layer. The output layer is omitted in this figure. The connection weight is written next to the edges and the bias is written on top of the hidden neurons. In neuron-based computing, connection weights are members of neurons rather than members of the connection matrix; so, whenever possible, the weights are drawn closer to its owners. A target function is written in terms of a set of subsets. For each subset in the target function, there is one hidden tree that can identify it and only it. The last hidden layer has one neuron for each subset, so the neural network can implement any target function.

Why locally Connected? A Time and Space Complexity Analysis

Time complexity measures how the running time of an algorithm grows as the size of its input increases. Space complexity measures the amount of memory or storage required by an algorithm relative to the size of its input. The implicit assumption here is that the comparison between a fully connected network and locally connected network is based on the fact that

the same target function can be identified by both. Let *d* be the number of input neurons; let h = {s_0, s_1, …, } be a target function; and, let |h| be the number of strings in set *h*. In the fully connected network, there are *d* neurons from the input layer, |h| neurons from the hidden layer, and one neuron from the output layer for a total of d + |h| + 1 neurons. The hidden layer has d*|h| connections and the output layer has |h| connections. For one pass of training, the time and space complexities are T = O (d*|h|) and S = O (d*|h|).

For bilocal networks, there are *d* neurons from the input layer, |h| * (d-1) neurons from the |h| hidden trees, and one neuron from the output layer for a total of d + |h| *(d-1) + 1 neurons. The number of hidden neurons is significantly higher, which is increased by a factor of O(d), from |h| to |h| * (d-1). There are also |h| binary trees, where each tree has 2 * d – 2 connections. The hidden layer has (2 * d – 2) * |h| connections, and the output layer has |h| connections. The number of connection weights is roughly doubled. This trade-off has the potential of improving time complexity at a minor cost of more neurons and connection weights. The space complexity is primarily determined by connection weights, not by the number of neurons, so the space complexity does not increase when the number of neurons is increased by a factor of O(d). Doubling the weights will also not change the space complexity, which measures the order of magnitude, and a constant of 2 will not change the space complexity. For one pass of training, both the time and space complexities are T' = O (d*|h|) and S' = O (d*|h|).

From the time and space complexity analyses, there are no advantages for the locally connected network; however, this is not true for the following reasons. First, the DNN itself attempts to localize the network by dividing the network into many layers; the deeper the network, the more locally connected the network will become. Second, the von Neumann bottleneck, which refers to the limit of computing systems that stems from the separation of the central processing unit (CPU) and RAM, is another problem. Training of large networks demands substantial hardware because:

1) Parameters: the scale of these models is immense and the memory requirements to store and process these parameters are significant, prompting the transition from CPU to GPUs and from GPU to IPU, TPU, and NPU (Brown et al., 2020). The computational complexity is primarily driven by the extensive matrix multiplications and gradient descent calculations involved in backpropagation, which require multiple passes through the entire network (Goodfellow, Bengio, & Courville, 2016).
2) Training data: training on vast datasets requires not only significant storage but also powerful computational resources to handle the iterative processes involved in training (Devlin, Chang, Lee & Toutanova, 2018).
3) Training: high throughput for data processing necessitates advanced storage systems and network infrastructure to efficiently feed data to the model. The distributed nature of training across multiple GPUs/TPUs adds further complexity (Rajbhandari, Rasley, Ruwase & He, 2020).
4) Power: high energy consumption can also be a problem (Jouppi et al., 2017). Simply speaking, it is impossible for the cache memory to hold so much data, so most of the time and power are consumed by moving data. It can be significantly helpful if the computations are basically local. This reduction of thrashing will not reduce the accuracy of the computation.

It is essential to emphasize the significance of incorporating locality in neural networks and its implications for computation efficiency. Using the earlier example in this section, in a fully connected network, there are *d* neurons from the input layer and |h| neurons from the hidden layer; the hidden layer has d*|h| connections. For one pass of training, the time and space complexities are T = O (d*|h|) and S = O (d*|h|). When both d and |h| are very large, the connection matrix (d x |h| dimension) is very large, and only a small portion of this matrix can be held in RAM. To complete a matrix multiplication, a portion of a large matrix is loaded into RAM, then removed from RAM to make room, only to find that it will need to be reloaded again. Increasingly, computation times are spent on moving data from one place to another. Assuming the same matrix will need to be reloaded R times on average, the mathematical time complexity of T = O (d*|h|), which assumes unlimited RAM, is actually T = O (d*|h|*R), where R is the average number of reloads for a large matrix. R is 1 only if the memory is as large as d*|h|, which is simply not the case for large matrices.

For bilocal networks, the number of hidden neurons is significantly higher, which is increased by a factor of O(d); however, the number of connection weights is roughly the same order of magnitude. It is the connection weights that determine the time and space complexities. The number of connection weights is roughly doubled, but there is no large matrix here so the data does not need to be loaded and unloaded over and over again. Why is there no connection matrix in a neuron-based computation? The connection weights are members of neurons. There are two computations: forward computations of neuron values and backward computations of weight updates.

1) When neuron values in the first hidden layer are calculated, they can be calculated one neuron at a time; this is because all weights of a neuron are properties of this neuron. To update a neuron value, the members of this neuron alone are enough to complete the neuron value calculation. When all of the neurons are updated in the first hidden layer, the process can be repeated for the second hidden layer, again one neuron at a time. Note that there is no matrix.
2) Similarly, when new weights are calculated, they can be calculated in such a way that only one neuron is used at a time; this is because the weight update training related to one neuron is based on all of the weights connected to this neuron in a backward direction. When

all of the neurons in the output layer are processed, one can repeat the process for the last hidden layer, again one neuron at a time, gradually moving backward. Note that there is again no matrix. For example, one can compute the responsibility of a neuron based on all the weights connected to this neuron in a backward direction. In both cases, only one-dimensional arrays are used because only one neuron is processed at a time and these arrays will be loaded into RAM only once. For one pass of training, the time complexity is T' = O (d*|h|), which can be significantly faster than T = O (d*|h|*R), in the case of fully connected neural networks, where R is the average number of reloads for a large matrix.

Incorporating locality in neural networks can increase computation efficiency by a factor of R. It is this factor of R that opens a discussion for exploring an approach of locally connected neural networks as an alternative to globally connected models. OpenAI's GPT-3, the architecture underlying ChatGPT-3, is one of the largest and most sophisticated language models developed with known size (Brown et al., 2020). The largest GPT-3 model, often referred to as GPT-3 175B, has 96 layers (transformer blocks) and 175 billion parameters. Each layer has an Attention block and a Feedforward Network.

The attention block has four 12,288 x 12,288 matrices, where three of the matrices will multiply (Vaswani et al., 2017). The Feedforward Network (FFN) has two 12,288 x 49,152 matrices. Together, these matrices give the majority of the 175 billion parameters. The number of parameters in GPT-4 is not officially disclosed by OpenAI, but it is expected that the cost of training ChatGPT-4 is an order of magnitude higher than ChatGPT-3 and the cost of training ChatGPT-5 will be an order of magnitude higher than ChatGPT-4 (Wall Street Journal, 2024). Therefore, it is important to increase computation efficiency.

In a locally connected network, the basic computation unit is a neuron rather than a connection matrix. In the extreme case of a bilocal network, a neuron value, a few weights, and a bias form the foundation of computation, irrespective of how large the network is. This is in contrast to the connection-matrix-based computation unit that grows with the network size. As a reference, biological neural networks are locally connected, where data movement is minimum and the number of neurons is large. The biggest difference between fully connected networks and bilocal networks is that one uses matrix-based computation and the other uses neuron-based computation; one uses matrices as a computation unit and the other uses neurons as computation units. As a reference, the neuroscience textbook by Kandel, Schwartz, and Jessell (2013) states that individual neurons in the human brain typically form between 1000 and 10,000 synaptic connections. The biological neural net has two features: it has a large number of neurons and is locally connected.

Discussion

Earlier, the authors made two assumptions for easy discussion: bilocal and 2^K input neurons. Now these assumptions will be removed.

Arbitrary number of input neurons:
To move from 2^K to an arbitrary number, the process is standard and well-known, such as binary search and merge sort. For example, let the input layer have 11 neurons:

[0,1,2,3,4,5,6,7,8,9,10]

Following the binary search or merge sort process, the division for an integer interval [a, b] is [a, m] and [m+1, b], where m = (a + b)/2 is an integer division. The division process then is:

[0,1,2,3,4,5,6,7,8,9,10]
[0,1,2,3,4,5], [6,7,8,9,10]
[0,1,2], [3,4,5], [6,7,8], [9,10]
[0,1], [2], [3,4], [5], [6,7], [8], [9,10]

Now there are some singleton neurons left in the input layer. A single neuron can be identified by a hidden neuron using the same rule noted previously.

N-ary tree:
N-ary tree is a tree in which a node can have at most N children. Binary trees are specific cases where N = 2. The binary connections are merely for easy discussion. By using an N-ary tree, all the restrictions that were imposed, for the sake of easy discussions, are removed. The rules allow one hidden neuron to identify arbitrary numbers of bits; therefore, all of the rules apply to the N-ary trees, which is: let *s* be a subset given by Equations 8 and 9, and assume that a hidden tree will identify *s*; the neurons in the first hidden layer then have weights and biases as follows:

set weight = L, for input neurons $\{j_0, j_1, j_2, \ldots\}$
set weight = – L, for all other input neurons
set bias = $-(|s| - 1) \cdot L$

Neurons in the rest of hidden layers have weights and biases determined by the above rule for identification of patterns: "11…1".

Universal Approximator with Two and Three Hidden Layers

For a given input number, *d*, and a given number of layers, there are numerous constructions, where the number of neurons in the hidden layers depends on the construction. Figure 5 gives an example of a single hidden tree for d = 8, two hidden layers, and the maximum localization construction (the number of edges is maximum). The figure shows the input layer, the first hidden layer, and the last hidden layer. The

output layer is omitted. Figure 6 gives an example of a single hidden tree for d = 8, two hidden layers, and the minimum localization construction (the number of edges is minimum). Again, the figure shows the input layer, the first hidden layer, and the last hidden layer. The output layer is omitted.

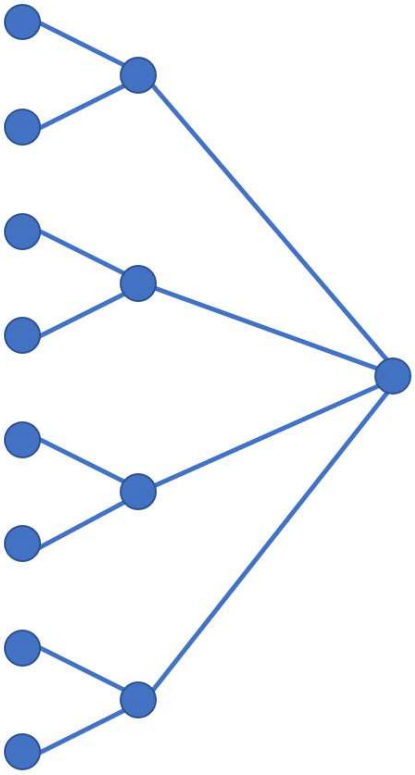

Figure 5. An example of maximum localization construction.

As *d* grows larger, the diagram gets harder to read, so a new notation will be introduced:

- Let the input neurons be labeled by "Input Layer: 0, 1, 2, …, 15";
- Let the neurons in the first hidden layer be labeled by "First Hidden Layer: 0, 1, 2, ….";
- Let the neurons in the second hidden layer be labeled by "Second Hidden Layer: 0, 1, 2, …";
- Let the neurons in the last hidden layer be labeled by "Last Hidden Layer: 0, 1, 2, …"; and,
- Let "[...]" be used to group neurons together to be identified by a neuron in the next layer.

For example, Input layer: [0,1] [2,3] means input neurons 0 and 1 will be identified by neuron 0 in the first hidden layer and input neurons 2 and 3 will be identified by neuron 1 in the first hidden layer. Under this notation, Figure 5 can be rewritten as:

Input layer: [0,1] [2,3] [4,5] [6,7]
First hidden layer: [0,1,2,3]
Last hidden layer: [0]

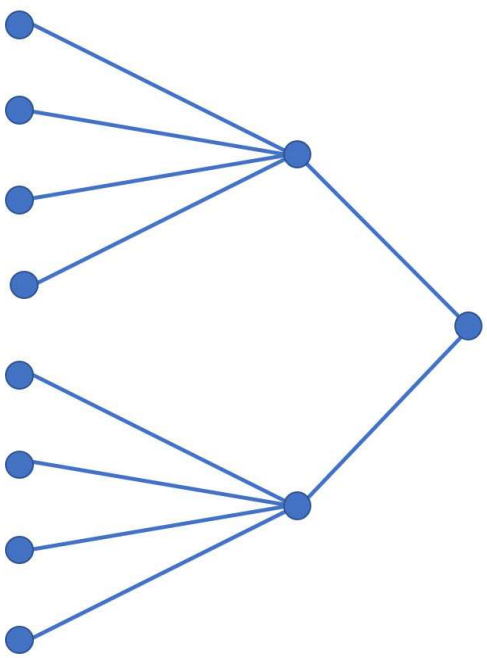

Figure 6. An example of minimum localization construction.

Figure 6 can be rewritten as:

Input layer: [0,1,2,3] [4,5,6,7]
First hidden layer: [0,1]
Last hidden layer: [0]

The following example will be more interesting, which is d = 16 and two hidden layers. In this example, let d = 16 and let a network have two hidden layers. The maximum localization construction looks like this:

Input layer: [0,1] [2,3] [4,5] [6,7][8,9][10,11][12,13][14,15]
First hidden layer: [0,1,2,3,4,5,6,7]
Last hidden layer: [0]

where,

- input neurons [0,1] are identified by neuron 0 of the first hidden layer,
- input neurons [2,3] are identified by neuron 1 of the first hidden layer,
- …, and
- first-hidden-layer neurons, [0, …, 7], are identified by neuron 0 of the last hidden layer.

The minimum localization looks like this:

Input layer: [0,1,2,3,4,5,6,7] [8,9,10,11,12,13,14,15]
First hidden layer: [0,1]
Last hidden layer: [0]

The intermediate localization looks like this:

Input layer: [0,1,2,3] [4,5,6,7] [8,9,10,11] [12,13,14,15]
First hidden layer: [0,1,2,3]
Last hidden layer: [0]

Clearly, there are many other constructions. As a comparison, with two hidden layers and minimum localization, it basically divides the original fully connected network into two networks, which reduces the weight connection matrix. With two hidden layers and maximum localization, it basically reduces the size of the original fully connected network by half by taking the average of two inputs and combining it into one input, which again reduces the connection matrix. In either case, it is a small deviation from the original network. As more and more layers are added, the difference between fully connected networks and locally connected networks will get bigger and bigger; eventually, it will transit from a matrix-based computation to a neuron-based computation. Universal approximators with three hidden layers can be constructed in a similar way.

Conclusions

In earlier work by the authors, they showed that an arbitrary binary target function can be effectively rewritten in terms of a set of strings, or a set of subsets, and that a single hidden neuron can identify and only identify a single string or a single subset; therefore, an arbitrary binary target function can be effectively rewritten in the form of a neural network with one hidden layer, thus proving that deep neural networks can effectively implement any target mappings. In this paper, the authors imposed locality on the neural network and showed that an arbitrary binary target function can be effectively rewritten in the form of a locally connected DNN, which can have many hidden layers. When locality is imposed on the network, the basic computation unit can be shifted to neurons rather than connection matrices. Continuous loading of batches of data from storage into memory to processing units can be significantly reduced. By imposing locality, the computation power of the DNN is not decreased, but it can reduce thrashing, thus significantly increasing computation speed.

Acknowledgments

This work was supported by the Department of Energy Minority Serving Institution Partnership Program (EM-MSIPP) managed by the Savannah River National Laboratory under BSRA contract TOA Number: 0000663608 and National Science Foundation under Award Number 2348805. The authors would like to thank Gina Porter for proofreading this paper.

References

Amari, S., Kurata, K., & Nagaoka, H. (1992). Information Geometry of Boltzmann Machine. *IEEE Trans., Neural Network, 3*(2), 260-271.

Anthropic. (2023). Claude: A Language Model for Conversational AI. https://www.anthropic.com/index/claude

Bengio, Y. (2009). Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning, 2*(1), 1-127. http://dx.doi.org/10.1561/2200000006

Bengio, Y., Courville, A., & Vincent, P. (2013). Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 35* (8), 1798-1828. doi:10.1109/tpami.2013.50 https://ieeexplore.ieee.org/document/6472238

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P. ... & Amodei, D. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems, 33*, 1877-1901.

Byrne, W. (1992). Alternating Minimization and Boltzmann Machine Learning. *IEEE Trans., Neural Network, 3*(4), 612-620.

Cheng, X., Li, Y., & Lu, J. (2019). Butterfly-Net: Optimal Function Representation Based on Convolutional Neural Networks. https://arxiv.org/pdf/1805.07451

Ciresan, D., Meier, U., & Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition*, 3642-3649. doi:10.1109/cvpr.2012.6248110 https://ieeexplore.ieee.org/document/6248110

Coursera. (2017). Coursera, Your Course to Success. https://www.coursera.org/

Cybenko, G. (1989). Approximation by Superposition of a sigmoid function. *Mathematics of Control, Signals, and System, 2*, 303-314.

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. https://doi.org/10.48550/arXiv.1810.04805

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

Han, S., Pool, J., Tran, J., & Dally, W. (2015). Learning both weights and connections for efficient neural networks. *Advances in Neural Information Processing Systems, 28*, 1135-1143.

Hinton, G. E., Osindero, S., & Teh, Y. (2006). A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation, 18*, 1527-1554.

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer Feedforward Networks are Universal Approximators. *Neural Networks, 2*, 359-366.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T. ... Adam, H. (2017). MobileNets: Efficient convolutional neural networks for mobile vision applications. https://doi.org/10.48550/arXiv.1704.04861

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R. ... Yoon, D. (2017). In-datacenter performance

analysis of a tensor processing unit. *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 1-12.
Kandel, E. R., Schwartz, J. H., & Jessell, T. M. (2013). *Principles of Neural Science* (5th ed.). McGraw-Hill Education.
Kubat, M. (2015). *An Introduction to Machine Learning.* (1st ed.). Springer.
Le Roux, N., & Bengio, Y. (2008). Representational Power of Restricted Boltzmann Machines and Deep Belief Networks. *Neural Computation, 20*(6), 1631-1649.
LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86*(11), 2278-2324.
LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep Learning. *Nature, 521*(7553), 436-444.
Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D. ... Wu, H. (2018). *Mixed precision training.* International Conference on Learning Representations. https://doi.org/10.48550/arXiv.1710.03740
Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems, 25*, 1097-1105.
Liu, Y. (1993). Image Compression Using Boltzmann Machines. *Proc. SPIE, 2032*, 103-117.
Liu, Y. (1995). Boltzmann Machine for Image Block Coding. *Proc. SPIE, 2424*, 434-447.
Liu, Y. (1997). Character and Image Recognition and Image Retrieval Using the Boltzmann Machine. *Proc. SPIE*, 3077, 706-715.
Liu, Y. (2002). Attrasoft Image Retrieval. US Patent, *7,773,800*. http://www.google.com/patents/US7773800
Liu, Y., & Wang, S. H. (2018). Completeness Problem of the Deep Neural Networks. *American Journal of Computational Mathematics, 8*, 184-196. https://doi.org/10.4236/ajcm.2018.82014
Liu, Y. (2018a). Linear Neurons and Their Learning Algorithms. *Journal of Computer Science and Information Technology, 6*(2), 1-14.
Liu, Y. (2018b). Square Neurons, Power Neurons, and Their Learning Algorithms. *American Journal of Computational Mathematics, 8*, 296-313. https://doi.org/10.4236/ajcm.2018.84024
Liu, Y., & Yousuf, A. (2020). Deep Neural Network and Universal Approximators. *International Journal of Modern Engineering, 20*(2), 45-50. https://ijme.us/issues/spring2020/X__IJME%20spring%202020%20v20%20n2.pdf#page=47
OpenAI. (2023). ChatGPT: Language Model for Dialogue Applications. https://openai.com/chatgpt
Rajbhandari, S., Rasley, J., Ruwase, O., & He, Y. (2020). *ZeRO: Memory Optimization Towards Training A Trillion Parameter Models*. https://doi.org/10.48550/arXiv.1910.02054
Schmidhuber, J. (2015). Deep Learning in Neural Networks: An Overview. *Neural Networks, 61*, 85-117. https://doi.org/10.1016/j.neunet.2014.09.003
Tan, M., & Le, Q. (2019). EfficientNet: Rethinking model scaling for convolutional neural networks. *Proceedings of the 36th International Conference on Machine Learning*, 6105-6114.
Theano. (2017). Theano 1.0. https://en.wikipedia.org/wiki/Theano_(software)
Tensorflow, (2017). Tensorflow. https://www.tensorflow.org
Torch, (2019). Torch, A scientific computing framework for LuaJIT. http://torch.ch/
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N. … Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 5998-6008.
Wall Street Journal. (2024). Eric Schmidt Walks Back Claim Google Is Behind on AI Because of Remote Work. https://www.wsj.com/tech/ai/google-eric-schmidt-ai-remote-work-stanford-f92f4ca5

Biographies

YING LIU is an Associate Professor of Computer Science Technology at Savannah State University. He received his master's degree and PhD in physics from Carnegie-Mellon University, and a Master's in Computer Science degree from the University of South Carolina. Dr. Liu has multiple Microsoft Certifications, including MCSE (Microsoft Certified System Engineer) and MCDBA (Database Administrator). He has published over 60 research papers, holds one patent, and over 30 software copyrights. Dr. Liu has extensive experience in software research and development in image recognition. Dr. Liu may be reached at liuy@savannahstate.edu

MAJID BAGHERI is an Assistant Professor of Civil Engineering Technology at Savannah State University. He received his PhD in Civil Engineering (environmental focus) from Missouri University of Science and Technology. Dr. Bagheri's expertise is in the modeling of environmental systems and environmental remediation using novel technologies such as artificial intelligence (AI) and machine learning (ML). He has developed several AI and ML models to improve the efficiency of treatment systems and assess the fate of environmental contaminants. Dr. Bagheri may be reached at bagherim@savannahstate.edu

ANTONIO VELAZQUEZ is an assistant professor at Savannah State University. He earned his BS in Civil Engineering (Structures and Construction) from Metropolitan Autonomous University (UAM-Mexico), a MEng in Structures from National Autonomous University of Mexico, a MSc in Computer Systems from National Polytechnic Institute (Mexico), a MSc in Wind Engineering from Northeastern University, a MSc in Fluid Dynamics (Hurricane Engineering) from Florida Institute of Technology, a PhD in Engineering Mechanics/Structures/MechEng from Michigan Technological University, and a post-doc in Machine-learning-based Engineering Education from Jackson State University. His research interests include

broad fields of engineering and computational mechanics. He is the author of several homemade CAD-like engineering software platforms developed at a commercial quality level. Dr. Velazquez has published research articles in international journals, conference proceedings, and workshops such as IWSHM, Journal of Sound and Vibration, Journal of Intelligent Material Systems and Structures, Journal of Computers (IEEE), and Journal of Engineering Structures. Dr. Velazquez may be reached at velazqueza@savannahstate.edu

ASAD YOUSUF is the department chairman for the Engineering Technology Department at Savannah State University. He received his BS in Electrical Engineering from NED University, MS in Electrical Engineering from the University of Cincinnati with emphasis on computer systems, and his doctorate from the University of Georgia. He has published several papers in technical journals and conference proceedings. He has years of experience in managing federally funded grants, having received over $2.5M in grant awards in the last few years. Dr. Yousuf may be reached at yousufa@savannahstate.edu