## Title

Applying Distributed Learning of Deep Neural Networks to Improve Their Classification Accuracy on Radio-Frequency Datasets



## Author

Schuette, Gregory Kyle Kenneth

## Publication Date

2020

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

# Applying Distributed Learning of Deep Neural Networks to Improve Their Classification Accuracy on Radio-Frequency Datasets

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Science in Chemistry

by

Gregory Kyle Kenneth Schuette

2020

ABSTRACT OF THE THESIS

Applying Distributed Learning of Deep Neural Networks to Improve Their Classification Accuracy on Radio-Frequency Datasets

by

Gregory Kyle Kenneth Schuette
Master of Science in Chemistry
University of California, Los Angeles, 2020
Professor Louis-Serge Bouchard, Chair

This thesis aims to improve on the current classification capabilities of deep neural networks on two types of radio-frequency data: radar and OFDM packets. In radar, applying neural networks to Automatic Target Recognition problems is a well-developed field, especially using the MSTAR database. However, existing state-of-the-art methods require precise preconditioning of radar data and are unsuited to applications with a large number of radar target classes. Therefore, we asked whether distributed learning can increase the generalizability and scalability of neural networks in these tasks. To test this, we applied distributed learning via Multi-Stage Training and a new network architecture, the Convolutional Multi-Stage Network, to provide a scalable, generalized treatment of radar data for more practical applications. This method was shown to outperform traditional neural network architectures on a new radar dataset. A similar approach was applied to the OFDM data with the goal of identifying specific radio-frequency transmitters for network security purposes. The task of identifying OFDM packet transmitters has previously been performed successfully, though with precise data collection methods. Data collection methods on a live network will likely include imperfect recording times, so we sought to improve network robustness to time-shifted OFDM packets. It was shown that the Convolutional Multi-Stage Network improved robustness to time-shifting of the radio-frequency data over the Multi-Stage Network, which

was the previous-best method. Simple preconditioning of the data using variations of the discrete wavelet transform further improved robustness to time-shifting of the radio-frequency data using both network architectures. These results are significant, as they provide a new avenue for applying neural networks to radio-frequency in difficult, real-world applications.

The thesis of Gregory Kyle Kenneth Schuette is approved.

Benjamin J. Schwartz

Justin R. Caram

Louis-Serge Bouchard, Committee Chair

University of California, Los Angeles

2020

# Table of Contents

# List of Figures

# Preface

First, I would like to thank Professor Louis Bouchard of the University of California, Los Angeles (UCLA) Department of Chemistry and Biochemistry. As my thesis advisor, he has continually supported me through my master's journey, teaching me how to perform academic research. He warmly accepted me to his laboratory and is always available and happy to provide guidance through difficulties.

I would also like to thank Dr. Khalid Youssef, a former graduate student and postdoctoral researcher in the Prof. Louis Bouchard Laboratory. A machine learning expert, Dr. Youssef helped guide my training in neural networks, the central topic in this thesis. Additionally, Dr. Youssef's work with Prof. Bouchard is cited heavily throughout the thesis, as it laid the foundation for my work. Kindly, he also offered feedback during the preparation of this thesis, increasing its quality.

I would also like to express my gratitude to Prof. Benjamin J. Schwartz and Prof. Justin R. Caram of the UCLA Department of Chemistry and Biochemistry. I first met these individuals when they instructed courses in which I was enrolled. Beyond the classroom, they have supported me with continued academic advice and generously accepted positions as committee members for my master's degree.

I would also like to thank Prof. Yung-Ya Lin of the UCLA Department of Chemistry and Biochemistry. My early research, not represented in this thesis, was performed with Prof. Lin. He introduced me to the world of research and is largely responsible for my interest in scientific computing.

Finally, I want to thank my parents for their endless support and encouragement throughout my academic journey. I would not be where I am without them.

# Curriculum Vitae

| | |
|---|---|
| 2015- | BS in Chemistry, Physical Chemistry Concentration, University of California, Los Angeles (UCLA). |
| | BS in Applied Mathematics, UCLA. |
| | MS in Chemistry, UCLA (Candidate) |
| 2019-2020 | Departmental Scholar, Department of Chemistry and Biochemistry, UCLA. |
| 2020- | PhD Student in Physical Chemistry, Massachusetts Institute of Technology |

# CHAPTER 1

# Introduction

The topic of this thesis is relatively non-traditional for a Chemistry degree. To account for this and hopefully guide future students from non-traditional backgrounds hoping to learn about machine learning, this thesis is written with the assumption that a reader has no prior knowledge regarding deep learning. This Introduction serves to provide a basic understanding of the concepts and mathematical underpinnings of deep learning as it relates to Multi-Stage Training (MST), a technique utilized and expanded upon in later sections. MST is a UCLA invention [2, 3].

*Classification* tasks are worked with throughout this thesis. The goal of these tasks is to use a neural network to interpret input data as belonging to some specified group. For example, a classic problem is to use a neural network to identify handwritten digits from 28x28 pixel black-and-white images; this is typically performed with the Modified National Institute of Standards and Technology (MNIST) dataset [4]. Tasks discussed in Chapter 2 considers identifying objects based on their radar signatures, and Chapter 3 considers identifying specific radio-frequency (RF) transmitters based on their fingerprints. Here, we discuss the progression to modern deep neural networks which enable these applications.

## 1.1 Artificial Neurons

Most deep neural networks consist of basic functional units known as *neurons*. These have a relatively long history and a well-defined mathematical form.

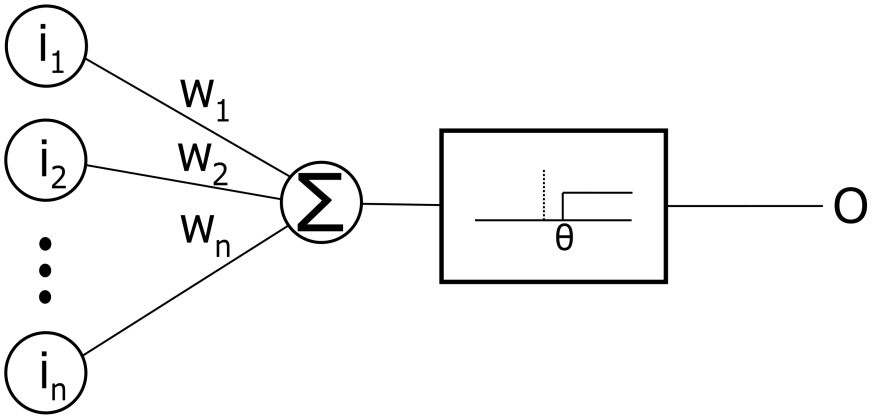


**Figure 1.1:** The McCulloch-Pitts Neuron is shown. A generalized number of inputs i are shown. Each is required to be a boolean value, i.e. 0 or 1. Each input $i_m$ is multiplied by the corresponding weight $w_m$. The products are then summed. The sum's value is compared to the threshold value $\theta$, determining whether the neuron's output, O, should be 0 or 1.

### 1.1.1 McCulloch-Pitts Neuron

True biological neurons are simultaneousy digital – either "fire" and propagate an electrical signal, or don't fire and halt an electrical signal – and analog, permitting a continous range of output signal amplitudes when firing occurs [5]. In an attempt to simulate these, the first artificial neuron was developed in 1943 by McCulloch and Pitts [6, 7]. In doing so, these researchers treated biological neurons as purely digital [6, 7]. Effectively, this is a decision-making process performed by a basic unit which the authors sought to replicate mathematically [7]. To do this, they developed the McCulloch-Pitts Neuron, shown in Fig. 1.1.

These neurons could output one of two values based on input data: a 1 or a 0 [8]. This is performed via a simple composite function, generally denoted $f(g(\vec{x}, \vec{w}))$ [8]. Here, $\vec{x}$ is the neuron's input represented by a vector of ones and zeros, i.e. $x_i \in \{0, 1\}\ \forall\ x_i \in \vec{x}$. $\vec{w}$ is a vector containing each of the neuron's *weights*, which are described further throughout Section 1.1. These vectors can be any length $N \in \mathbb{N}$, though should be of equal length [9]. Each element of $\vec{x}$ is limited to being a one or a zero, as these represent each prior neuron either firing fully or not firing at all, respectively [8]. $g(\vec{x}, \vec{w})$ represents the function $\sum_{i=1}^{N} w_i * x_i$,

where each $w_i \in \mathbb{R}$ is known a sfingle weight [8]. For the McCulloch-Pitts Neuron, these weights are chosen manually [6]. Another component of the equation, the *threshold value* $\Theta$, is chosen for the function $f$ [8]. Using the threshold value, the function $f$ is a simple piecewise function, and the artificial neuron is as follows [6, 9]:

$$f(g(\vec{x}, \vec{w})) = \begin{cases} 1 & \text{if } g(\vec{x}, \vec{w}) \geq \Theta \\ 0 & \text{Otherwise} \end{cases} \tag{1.1}$$
$$\text{where } x_i \in \{0, 1\} \ \forall \ x_i \in \vec{x}$$

In other words, the composition $f \circ g$ takes some input $\vec{x}$ and list of weights $\vec{w}$ and decides whether or not the neuron should fire. This can be understood as a *boolean* expression, or a true or false statement. In practice, this allows a neuron to distinguish between two types of inputs based on the output value of $g$ when proper weights are chosen [6]. Weights, threshold values, and other modifiable values in a network are known as the *parameters* of the network.

One simple example of this is preparing a McCulloch-Pitts Neuron to fire whenever fewer than five of ten input neurons had fired. For this, we can set all ten weights equal to -1. We also set the threshold value equal to -4. We then have:

$$f(g(\vec{x}, \vec{w})) = \begin{cases} 1 & \text{if } -\sum_{i=1}^{10} x_i \geq -4 \\ 0 & \text{else} \end{cases} \tag{1.2}$$

In this function, for every input of 1 ("prior neuron $i$ fired"), the value of $g$ decreases by 1. For every input of 0 ("prior neuron $i$ did not fire"), the value of $g$ is unchanged. Thus, the output value of $g(\vec{x}, \vec{w})$ is a negative tally of how many input neurons fired. If five or more neurons fired, then the value is less than or equal to -5. Thus, the "true" value for "fewer than five of ten input neurons had fired" should be outputted whenever the output of $g(\vec{x}, \vec{w})$ is greater than or equal to -4. This "true" value corresponds to the output of 1, while the "false" value corresponds to 0. The conversion of the output of $g$ to the binary output is performed by $f$.

More practically, several fundamental boolean operations can be performed by a McCulloch-Pitts Neuron [8]. These are: AND, OR, AND NOT, NOR, NAND, and NOT functions [8]. The McCulloch-Pitts Neuron for each of these is shown below with a simple corresponding truth table for a small number of inputs.

AND:

$$f(g(\vec{x},\vec{w})) = \begin{cases} 1 & \text{if } \sum_{i=1}^{N} x_i \geq \text{N} \\ 0 & \text{Otherwise} \end{cases} \tag{1.3}$$

| $x_1$ | $x_2$ | $g(\vec{x},\vec{w})$ | $f(g(\vec{x},\vec{w}))$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 2 | 1 |

OR:

$$f(g(\vec{x},\vec{w})) = \begin{cases} 1 & \text{if } \sum_{i=1}^{N} x_i \geq 1 \\ 0 & \text{Otherwise} \end{cases} \tag{1.4}$$

| $x_1$ | $x_2$ | $g(\vec{x},\vec{w})$ | $f(g(\vec{x},\vec{w}))$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 2 | 1 |

AND NOT:

$$f(g(\vec{x},\vec{w})) = \begin{cases} 1 & \text{if } x_1 - x_2 \geq 1 \\ 0 & \text{Otherwise} \end{cases} \tag{1.5}$$

| $x_1$ | $x_2$ | $g(\vec{x},\vec{w})$ | $f(g(\vec{x},\vec{w}))$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | -1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |

NOR:

$$f(g(\vec{x},\vec{w})) = \begin{cases} 1 & \text{if } -\sum_{i=1}^{N} x_i \geq 0 \\ 0 & \text{Otherwise} \end{cases} \tag{1.6}$$

| $x_1$ | $x_2$ | $g(\vec{x},\vec{w})$ | $f(g(\vec{x},\vec{w}))$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 1 | -1 | 0 |
| 1 | 0 | -1 | 0 |
| 1 | 1 | -2 | 0 |

NAND:

$$f(g(\vec{x},\vec{w})) = \begin{cases} 1 & \text{if } -\sum_{i=1}^{N} x_i > -\text{N} \\ 0 & \text{Otherwise} \end{cases} \tag{1.7}$$

| $x_1$ | $x_2$ | $g(\vec{x}, \vec{w})$ | $f(g(\vec{x}, \vec{w}))$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 1 | -1 | 1 |
| 1 | 0 | -1 | 1 |
| 1 | 1 | -2 | 0 |

NOT:

$$f(g(x, w)) = \begin{cases} 1 & \text{if } -x \geq 0 \\ 0 & \text{Otherwise} \end{cases} \tag{1.8}$$

| $x$ | $g(\vec{x}, \vec{w})$ | $f(g(\vec{x}, \vec{w}))$ |
|---|---|---|
| 0 | 0 | 1 |
| 1 | -1 | 0 |

While novel, this artificial neuron was extremely limited in function [6]. Furthermore, the necessity for a human operator to set all parameters greatly limited the purpose of these neurons in applications outside simple boolean operations [6]. Even here, it fails for use in all operations, as the XOR function cannot be represented by the McCulloch-Pitts Neuron [6]. The XOR function can be written as "either $x_1$ or $x_2$, but *not more* and *not less* than one of these." Thus, other researchers saught to develop these neurons for more general use [6, 8].

### 1.1.2 Perceptrons and ADALINE

Notably, Frank Rosenblatt developed the *classical perceptron* in 1958 [6, 8]. This artificial neuron allowed the input of non-boolean values. In other words, the artificial neuron's input is permitted to be $\vec{x} \in \mathbb{R}^N$ [6, 10]. Thus, the classical perceptron can be represented by the

function [6, 10]:

$$f(g(\vec{x},\vec{w})) = \begin{cases} 1 & \text{if } \sum_{i=1}^{N} w_i * x_i \geq \Theta \\ 0 & \text{else} \end{cases} \tag{1.9}$$
$$\text{where } \vec{x} \in \mathbb{R}^{\mathrm{N}}$$

where $w_i$ values are not required to be manually selected. Instead, the classical perceptron introduced the ability to "learn," or be algorithmically trained by data [6, 10]. This training process is further discussed in Section 1.3. With the introduction of classical perceptrons, a new convention came to prominence: The threshold value is subtracted from both sides, and the new function is compared to zero to determine the proper boolean output [8]. Furthermore, this parameter's name was changed from a *threshold value* to a *bias*, and its symbolic representation became $w_0$ [8]. This yields:

$$f(g(\vec{x},\vec{w})) = \begin{cases} 1 & \text{if } (\sum_{i=1}^{N} w_i * x_i) - w_0 \geq 0 \\ 0 & \text{else} \end{cases} \tag{1.10}$$
$$\text{where } \vec{x} \in \mathbb{R}^{\mathrm{N}}$$

In 1960, the *adaptive linear element* (ADALINE) was introduced [6, 10]. This improvement took the classical perceptron and modified its output; the ADALINE simply returns the value $g(\vec{x},\vec{w})$, providing a continous output rather than a boolean value [6, 8, 10]. This alters the connection between artificial and biological neurons, as the ADALINE's output can no longer be described as the firing (or lack of firing) of a neuron. Despite this, the term "artificial neuron" is still used to describe this mathematical object [6]. The ADALINE is represented by:

$$g(\vec{x},\vec{w}) = \sum_{i=1}^{N} w_i * x_i - w_0 \tag{1.11}$$
$$\text{where } \vec{x} \in \mathbb{R}^{\mathrm{N}}$$

### 1.1.3 Modern Neurons and Nodes

For the purposes of this thesis, ADALINE is the final relevant development to deep learning from deep learning's first wave of popularity, which ended in 1969 [6]. We should note that each artificial neuron discussed so far is a *linear model* [6]. In other words, fixing all values except one individual input $x_i$, weight $w_j$, or bias $w_0$ forms a function which is a line when plotted *prior* to thresholding. This, in fact, caused the end of the first wave of popularity, as critics showed that this greatly limited their application [6]. As discussed in section 1.1.1, the McCulloch-Pitts Neuron cannot learn the XOR function. In fact, this is true for all linear models [6, 11]. Studies throughout the 1960s exposed further weaknesses, including that parity – whether an odd or even number of inputs is activated – could not be solved by a single-layer perceptron, and the figure-ground problem – being able to distinguish relevant information from a background – could only be solved with impractically large networks [11].

To solve these issues, we can introduce nonlinearities via *activation functions*, also known as *transfer functions* [12]. These functions are applied to a neuron's output and can be any non-linear function mapping from the real number line (i.e. all possible outputs of the ADALINE neuron) to some other set of numbers [12]. The most common activation function today is the *rectified linear unit* (ReLU), which can be written simply as $\max(0, g(x))$ where $g(x)$ is the output of the ADALINE artificial neuron [6]. Despite the popularity of ReLU, the primary transfer function used in this thesis will be the hyperbolic tangent function, or [12]:

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{1.12}$$

A plot of the hyperbolic tangent function (Eq. 1.12), ReLU function (Eq. 1.13), sine function (Eq. 1.14), and softplus function (Eq. 1.15) are shown in Fig. 1.2. The softplus functions can be found at [13]. Using sine as a transfer function is a relatively new technique yielding promising results in image-processing problems [14]. Networks utilizing sine as a transfer

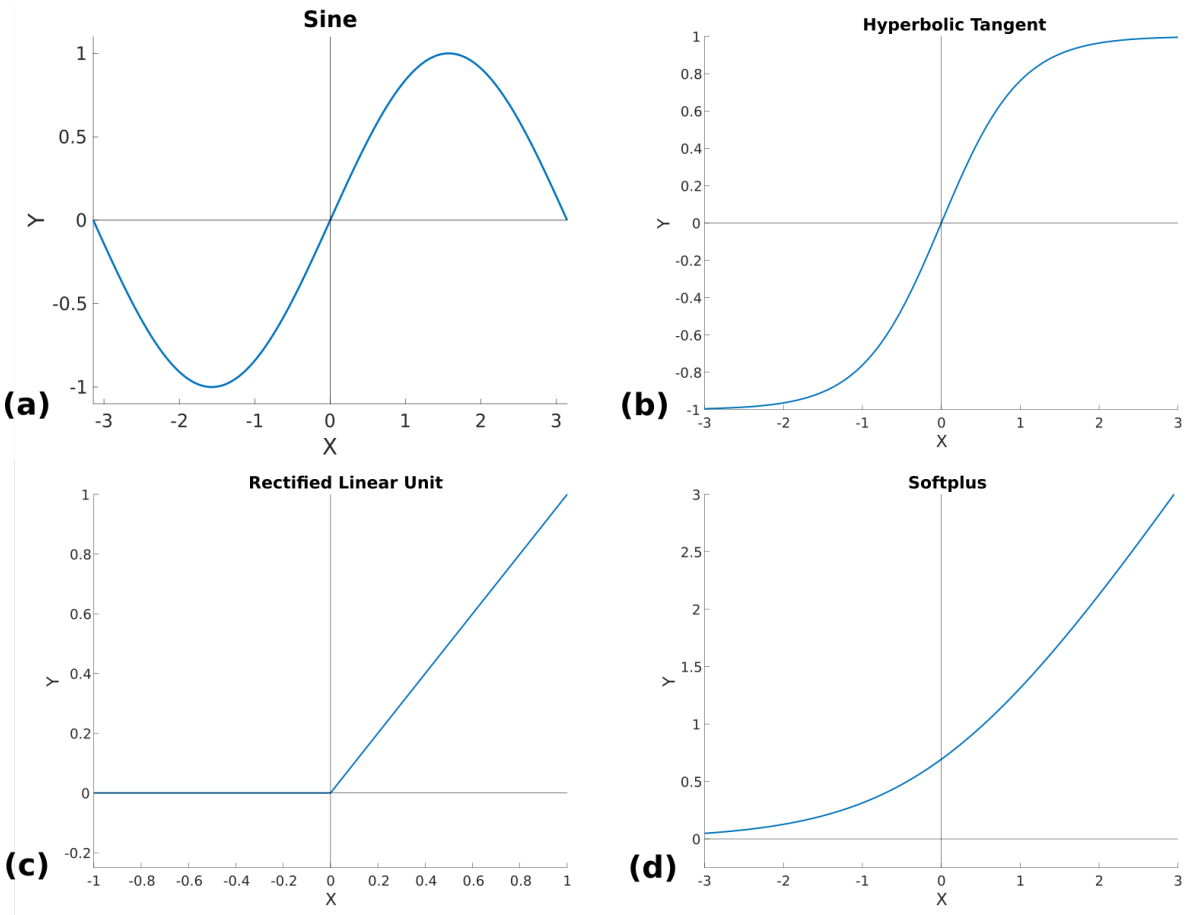


**Figure 1.2:** Four common transfer functions are shown. (a) The sinusoidal transfer function is plotted. This function goes $\mathbb{R} \to [-1, 1]$. (b) The hyperbolic transfer function is plotted. This function goes $\mathbb{R} \to (-1, 1)$. (c) The rectified linear unit (ReLU) is shown. The ReLU transfer function goes $\mathbb{R} \to [0, \infty)$. (d) The softplus function is plotted. This continuous, smooth function approximates the ReLU transfer function. The softplus function goes $\mathbb{R} \to (0, \infty)$.

function are known as *sinusoidal representation networks* (SIREN) [14].

$$f(x) = \max(0, x) \tag{1.13}$$

$$f(x) = \tag{1.14}$$

$$f(x) = \ln(1 + e^x) \tag{1.15}$$

This combined unit of an ADALINE artificial neuron and a transfer function, sometimes called a *node* [6, 12], will be used as the basic building block of all networks throughout the remainder of this thesis. Thus, our basic functional unit can be represented as:

$$\text{output } = f(\sum_{i=1}^{N} w_i * x_i - w_0) \tag{1.16}$$

Here, $\vec{x} \in \mathbb{R}^N$ is the node's input; $w_i$ for $i = 0$ and $i \in \{1, ..., N\}$ are the node's bias and weights, respectively; and $f$ is the transfer function.

For the remainder of this thesis, "neuron" will refer to the functional unit described in Eq. 1.16. This version of the neuron is used in many neural networks, several of which are discussed further in Section 1.2.

## 1.2 Neural Networks

The first *neural network* was described in 1959 by Bernard Widrow and Marcian Hoff [15]. This network, known as MADALINE or "Many ADALINE", consisted of several sequental layers of ADALINE neurons; each neuron is connected to every neuron in both the preceding and succeeding layers [16]. This is considered a *fully-connected network*, whose modern form is described in Section 1.2.1.

Later, in the late 1980s and early 1990s, *convolutional neural networks* were introduced. These networks greatly expanded the capabilities of neural networks, as discussed in Section 1.2.2.

### 1.2.1 Fully-Connected Neural Networks

*Fully-connected, feed-forward neural networks* (FCN) are networks with distinct and subsequent *layers* [17]. A layer can be visualized as a vector of neurons while the network can be visualized as a series of layers whose neurons are connected via mathematical operation; this is shown in Fig. 1.3. Each layer's output is a vector where each element is the output

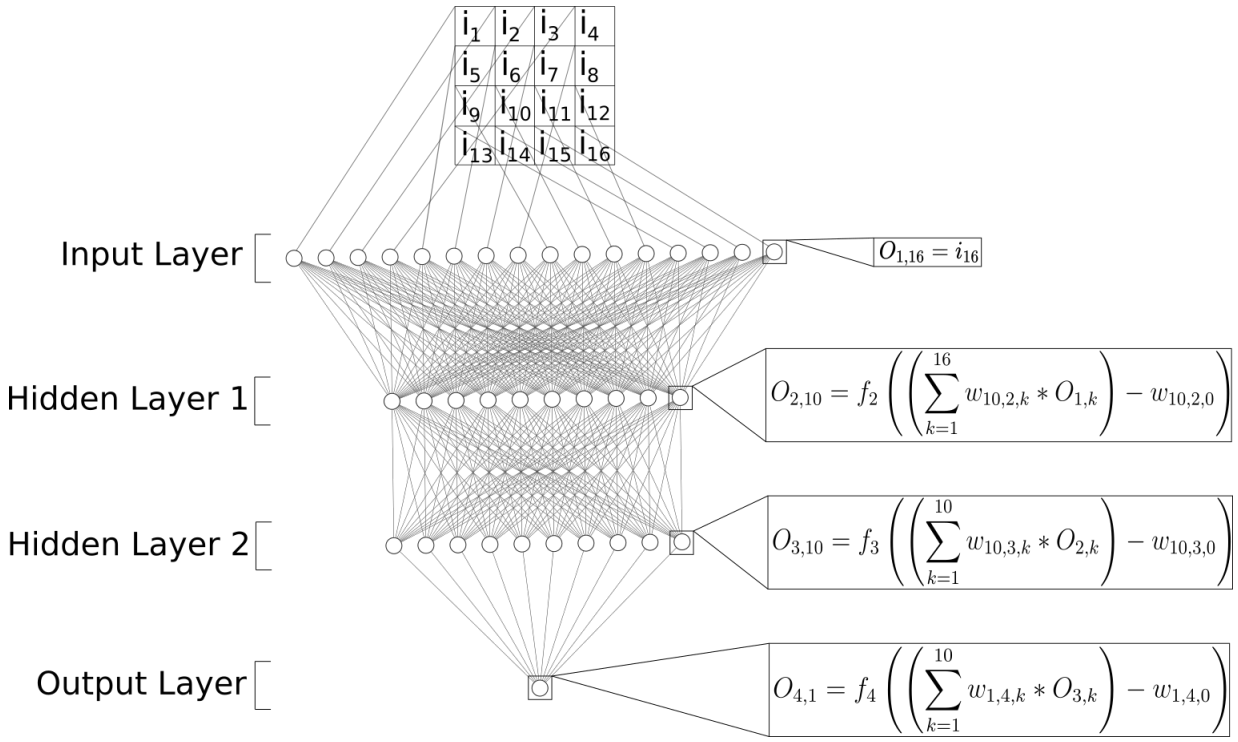


**Figure 1.3:** A sample FCN is shown. This has an input size of 16, two hidden layers with ten neurons each, and an output layer of size 1. Note that this is also the structure of an MLP, which is discussed further in Section 1.4. The input layer takes the two-dimensional input and vectorizes it. Each hidden layer operates on the output from the layer immediately preceding it. The output layer operates on the output of the final hidden layer. The specific operation at one node per layer is shown to the right. $f_2$, $f_3$, and $f_4$ refer to the transfer function at the second, third, and fourth overall layer, respectively. All outputs at a given layer are placed in a vector, as shown in Eq. 1.18. This figure was produced in part using [1].

of one neuron in the layer. This is shown in Eq. 1.18. Fundamentally, there are three types of layers.

The first layer, known as the *input layer*, is directly provided the data input by a user [6, 18]. The input to each subsequent layer is taken as the output from the layer immediately preceding it [6, 18]. The passing of data from one layer to the next continues until the final layer, known as the *output layer*, is reached [6, 18]. The output layer provides the final output of the FCN. Any layer that is neither the input nor the output layer is known as a *hidden layer* because both its input and output remain unseen by the user; these values are only transiently produced during network operation [6, 18].

The input layer contains one neuron per element of the network's input [18]. The output value for each of these neurons corresponds exactly to the value of one element of the original input, i.e. each neuron has a weight of one for a single element of the input and zeros for all else (and no two neurons have a weight of one for the same input element), a linear activation function of slope 1, and a bias of 0 [18]. More simply put, consider that each element of the input is uniquely labeled numerically, and each input neuron is labeled in the same manner. Then the weights for each neuron are:

$$w_{nm} = \begin{cases} 1 & \text{if } n = m \\ 0 & \text{Otherwise} \end{cases} \tag{1.17}$$

where $n$ is the neuron's label, and $m$ is the label of an element of the input. To simplify notation and calculations through the rest of the network, the input layer's output is placed in a vector with each entry corresponding to one neuron in the input layer. In other words, the input layer serves to vectorize the network's input data [18].

For example, if a network's input consists of matrix $\mathrm{B} \in \mathbb{R}^{\mathrm{NxM}}$, then the input layer contains $\mathrm{N} * \mathrm{M}$ neurons. A bijective labeling scheme is chosen such that each element of the input matrix contains a unique value $m \in \{1, ..., \mathrm{N} * \mathrm{M}\}$. This could be that element $b_{kl} \in B$ is relabeled $i_{(l-1)*N+k}$. Then, for the arbitrarily-chosen $n^{\mathrm{th}}$ neuron in the input layer, we see that the neuron's output is simply $i_n$, or the $n^{\mathrm{th}}$ element of the input matrix.

A user is permitted to choose the number of hidden layers when designing an FCN [6, 18]. While therefore not required, at least one hidden layer is generally included in a network [6, 18]. A user has more choice when designing their network, as any number of neurons may be placed in a hidden layer [6, 18]. The choices a user can make about a network's architecture are known as *hyperparameters* [18]. In this thesis, each neuron within a given layer will utilize the same activation function.

Continuing our prior example, choose the activation function $f$ and $\mathrm{N_{h1}}$ neurons for the first hidden layer. Recall that each neuron's input is the full vector produced by the prior layer. Thus, the output of a hidden layer is written:

$$\vec{o_2} = \begin{bmatrix} f(\sum_{i=1}^{\mathrm{N*M}} w_{1,2,i} * o_{1,i} - w_{1,2,0}) \\ \vdots \\ f(\sum_{i=1}^{N*M} w_{\mathrm{N_{h1}},2,i} * o_{1,i} - w_{\mathrm{N_{h1}},2,0}) \end{bmatrix} \tag{1.18}$$

where $w_{m,n,i}$ indicates the $i^{\text{th}}$ weight corresponding to the $m^{\text{th}}$ neuron in the $n^{\text{th}}$ layer; $w_{m,n,0}$ represents the the $m^{\text{th}}$ neuron's bias in the $n^{\text{th}}$ layer; and $o_{m,n}$ represents the $n^{\text{th}}$ output of the $m^{\text{th}}$ layer.

To simplify notation and decrease the difficulty of programming neural networks, Eq. 1.18 is converted to its matrix expression. The layer is also generalized so that the following is valid for any hidden layer:

$$\vec{o_j} = f\left( \begin{bmatrix} w_{1,j,1} & \dots & w_{1,j,\mathrm{N_{h(j-1)}}} \\ \vdots & \ddots & \vdots \\ w_{\mathrm{N_{hj}},j,1} & \dots & w_{\mathrm{N_{hj}},j,\mathrm{N_{h(j-1)}}} \end{bmatrix} \begin{bmatrix} o_{j-1,1} \\ \vdots \\ o_{j-1,\mathrm{N_{h(j-1)}}} \end{bmatrix} - \begin{bmatrix} w_{1,j,0} \\ \vdots \\ w_{\mathrm{N_{hj}},j,0} \end{bmatrix} \right) \tag{1.19}$$

In Eq. 1.19, $j$ represents the overall layer index starting with input layer numbered as 1. For example, the input layer would have a $j$ value of 1, the first hidden layer would have a $j$ value of 2, etc. $\vec{o_j}$ represents the output of the $j^{\text{th}}$ layer. $N_{hj}$ represents the number of neurons in layer $j$.

As stated, a user can select any number of hidden layers, each with any number of neurons. This choice of hyperparameters typically depends on the intended application of the network. For example, large networks can learn to identify objects from complicated datasets, but smaller networks are less prone to "memorizing" the training set which helps prevent overfitting [19]. Regardless of specific internal architecture, however, the output from the final hidden layer is treated as the input to the output layer.

The mathematics of an output layer are very similar to the hidden layers. However, there is the added 'restriction' that the number of neurons in the output layer must be equal to

the number of outputs of the network [6, 18]. Thus, Eq. 1.19 only needs minor adjustments to represent the output layer:

$$\vec{o} = f\left(\begin{bmatrix} w_{1,j,1} & \dots & w_{1,j,N_i} \\ \vdots & \ddots & \vdots \\ w_{N_o,j,1} & \dots & w_{N_o,j,N_i} \end{bmatrix} \begin{bmatrix} o_{j-1,1} \\ \vdots \\ o_{j-1,N_i} \end{bmatrix} - \begin{bmatrix} w_{1,j,0} \\ \vdots \\ w_{N_o,j,0} \end{bmatrix}\right) \tag{1.20}$$

In Eq. 1.20, $N_o$ represents the number of outputs, and $\vec{o}$ represents the final network output.

### 1.2.2 Convolutional Neural Networks

*Convolutional Neural Networks* (CNN) are a newer type of neural network. Yann LeCun was the main developer of these networks beginning in 1989 [6]. These networks effectively analyze data whose positioning is important [6]. This includes analyzing images – an exceptionally difficult task with fully-connected networks – and time-series data [6, 20]. The unique characteristic of CNNs is that a convolution operation is performed in at least one layer [6]. Compare this to the FCNs in Section 1.2.1 which only use standard matrix multiplication. Convolutions may be performed on continuous data, though the discrete form is simpler and all that is relevant for CNNs. Furthermore, most CNNs utilize the discrete *cross-correlation* operation rather than a true convolution [6]. Thus, we focus on the discrete, two-dimensional cross-correlation operation here. Following convention, however, this operation will be referred to as a convolution. Eq. 1.21 provides a single output value during this process, where Eq. 1.21 is iterated to yield multiple outputs [6]. Multiple output values yield a two-dimensional output as this operation is iterated.

$$S(i,j) = (K * I)(i,j) = \sum_{n=1}^{N}\sum_{m=1}^{M} I(i+m, j+n)\, K(m,n) \tag{1.21}$$

Here, the input $I$ is two-dimensional. $S(i,j)$ represents the output when the two-dimensional kernel $K$ operates on $I$ at the $(i,j)^{\text{th}}$ position. $K$ is of dimension $N$x$M$, which is

generally smaller than the input's size and can never be larger. The sum indicates element-by-element multiplication of $K$ on the rectangular subset of $I$ beginning with indices ($i$,$j$) and ending with indices($i + M$,$j + N$), followed by a sum of each resulting value [21].

Convolutional layers require the use of objects known as *kernels*, which can operate over multi-dimensional data [6, 21]. For two-dimensional convolutions, a kernel is a two-dimensional matrix of weights [21]. As stated before, kernels perform element-by-element multiplication on a subset of the input data [21]. These products are then summed [21]. This sum represents one output of the convolutional layer [6, 21]. When the layer's input is two-dimensional, this value represents a single output from the convolutional layer [21]. A kernel is typically chosen to be smaller than the two-dimensional data on which it is operated to enable different portions of the input to be analyzed by the filter in different positions. Thus, to operate on all of the input data, the kernel is progressively shifted – or *convolved* – across the two-dimensional subsection of the data, and the same operation shown in Eq. 1.21 is performed at each position at which the kernel is placed [6, 21]. The outputs from each of these operations are placed in a grid, with each output placed in a location corresponding to the kernel's relative position when calculated [21]. This convolution process is illustrated Fig. 1.5.

The rate at which the kernel is convolved across the input data is called its *stride*. The stride may vary in different directions, though it represents how many "steps" the filter is shifted between operations. Here, a step size of 1 is considered moving one column right or one row down for movements in the x- or y-direction, respectively. Effectively, this implies that after Eq. 1.21 is performed in the $(1,1)^{\text{th}}$ position, the same operation is performed in the $(1+a,1)^{\text{th}}$ position, where $a$ is the step size in the x-direction. This is performed sequentially, with strides taken in the x-direction until the kernel leaves the input's bounds. Once this occurs, a stride is taken in the y-direction, and the kernel is placed back on the left side of the input, so that the operation in Eq. 1.21 is performed at the $(1,1 + b)^{\text{th}}$ position, where $b$ is the step size in the y-direction. In the y-position $1 + b$, the kernel is again convolved in the x-direction until it leaves the bounds, at which point a second step is taken in the

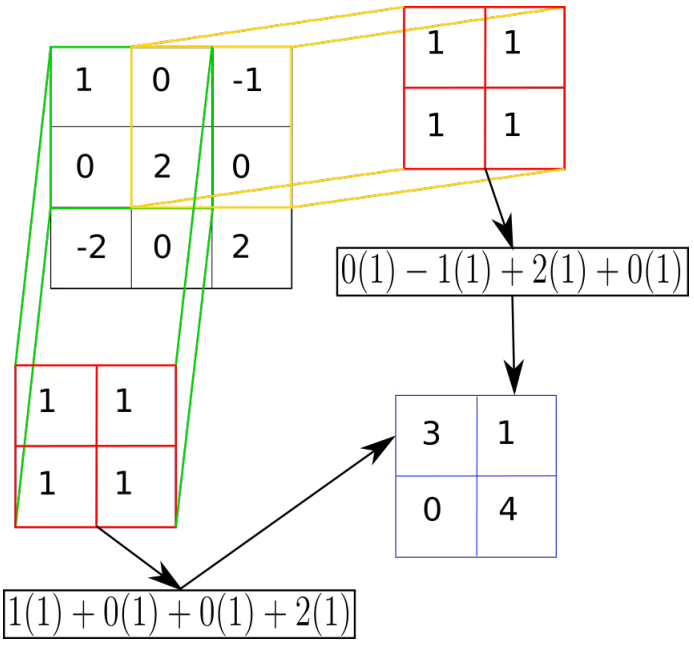


**Figure 1.4:** A two-dimensional kernel is shown in red operating on a two-dimensional input. The operation from Eq. 1.21 is shown occuring at two different indices on the input. These correspond to the green and yellow subsets of the input matrix. In this example, strides of length 1 are taken in both the x- and y-directions. With 3x3 input and 2x2 kernel, this results in an output of size 2x2, as there are two possible positions in both the x- and y-directions. Each output value is placed in the two-dimensional output, shown in blue, in the proper relative positions.

y-direction. The outputs at each unique position are placed in a two-dimensional grid. Each output is placed in the $(k,l)^{\text{th}}$ position, where $k$ steps were taken in the x-direction and $l$ steps were taken in the y-direction to produce the index at which Eq. 1.21 was performed. with of outputs, and this grid is a slice of the layer's output. A diagram of this process can be seen in Fig. 1.4.

If the convolutional layer's input is three-dimensional and two-dimensional convolutions are still desired, *filters* are typically created [21]. In this case, a filter is a three-dimensional matrix composed of multiple kernels concatenated in the z-direction. Each kernel is of equal size and, in general, the number of kernels composing the filter is chosen to be the depth of the input – also known as the number of channels [21]. In other words, when the filter is applied to a three-dimensional input, the filter will be convolved in the x- and y-directions, though convolutions will not be performed in the z-direction because the filter and input data

are equal in size in this dimension. The products from element-by-element multiplication between the filter and one subsection of the input are summed, and a single bias may be subtracted [6, 21]. This yields a two-dimensional output, analagous to the kernel output description shown in Fig. 1.4 [21]. If multiple filters are used, their outputs are concatenated in the final dimension to produce a three-dimensional input to the next layer [21]. This is shown in Fig. 1.5. Also, Eq. 1.22 represents this operation for one filter location, analagous to Eq. 1.21

$$S(i,j) = (F * I)(i,j) = \sum_{n=1}^{N} \sum_{m=1}^{M} \sum_{l=1}^{L} I(i+m, j+n, l)\, F(m,n,l) - b \tag{1.22}$$

Here, $F$ represents a filter composed of $L$ kernels, each of size $N$x$M$. The input is of size $L$ in the third dimension. $b$ represents the bias, which is often omitted in convolutional layers. All other variables are as described below Eq. 1.21.

Recall that this description of kernels and filters refers specifically to the two-dimensional convolution case. More generally, n-dimensional convolutions involve an n-dimensional kernel and (n+1)-dimensional filter [21].

CNNs can be customized in many ways, with far more features than are discussed here, though expanding on this is best left to outside resources. For this thesis, it is more important to understand what these convolutional layers enable.

First, if filters contain more than one weight and the input data's dimension sizes are not manipulated, then the output of a given layer will be smaller than the input. Notably, this is done with generally fewer parameters than an FCN layer [22]. With effective weights, this can serve to extract relevant features for multiple applications, including classification tasks [21, 23]. This all implies that data can be decreased in size with a relatively lightweight layer, all while maintaining information about the important features in the data [22].

Another imporant quality is that CNNs are able to learn the spatial dependencies of features in multi-dimensional objects [22]. Through various methods, these can be made to be shift- and rotational-invariant [22]. This shift-invariance property enables effective

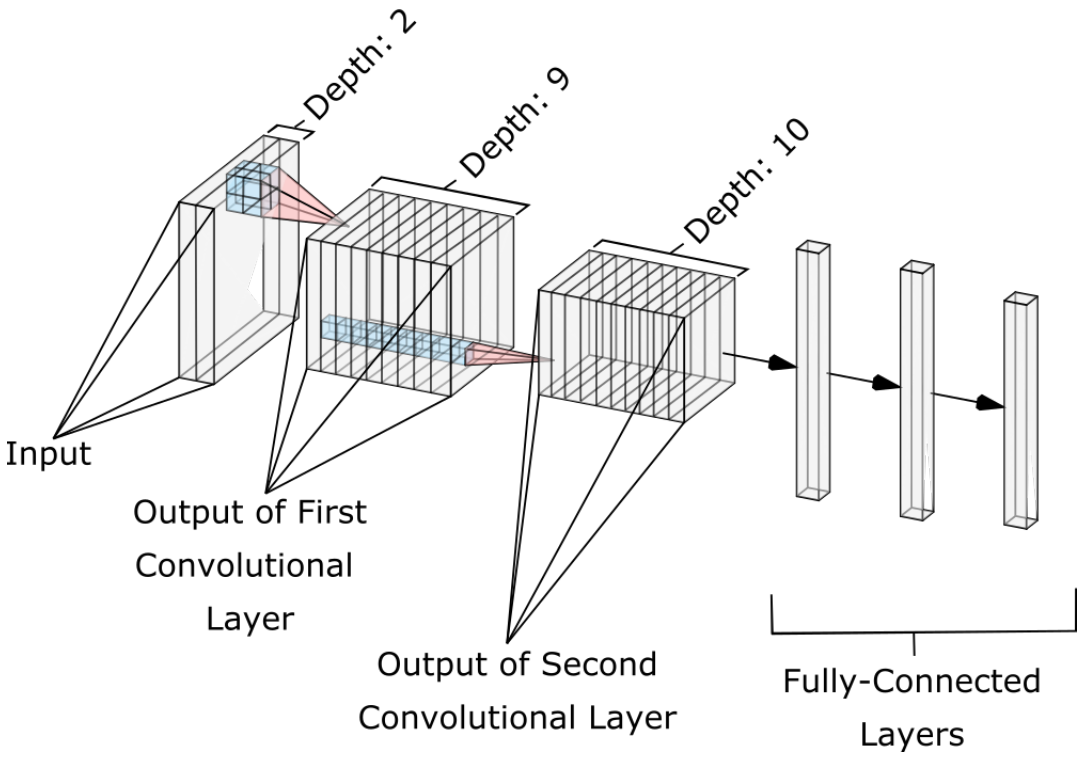


**Figure 1.5:** A CNN with two convolutional layers and a three-dimensional input is shown. This three-dimensional input can be considered a two-dimensional object with two channels, providing a depth of 2. Within each rectangular prism, a unit cell (1x1x1, where length 1 can be seen in the "depth" dimension) represents an individual value. Two filters are shown in blue, one per convolutional layer. Each filter is composed of multiple kernels stacked in the third dimension (depth). The number of kernels matches the depth of the convolutional layers' input, i.e. filters are composed of two stacked kernels in the first convolutional layer, and filters are composed of nine stacked kernels in the second convolutional layer. As such, two-dimensional convolutions are performed at each convolutional layer as the three-dimensional filter shifts to all available positions on the layer's input. This produces a two-dimensional output for each filter, represented as a slice in the convolutional layers' outputs. Multiple filters of equal sizes (not shown) produce multiple two-dimensional outputs, which are stacked to produce the three-dimensional convolutional layer outputs. The output of the second convolutional layer is vectorized to be used in a series of fully-connected layers at the end of the network. The vertical columns in the fully-connected layers represent the vectorized data fed into each fully-connected layer. This figure was produced in part using [1].

learning of time-shifted data, as discussed in Chapter 3.

## 1.3 Training Methods

Both FCNs and CNNs are powerful tools if *parameters* that allow a network to perform its intended function effectively are known, where parameters refer to the weights and biases of a network. However, networks that perform non-trivial tasks frequently contain thousands or millions of parameters [6, 24]. Furthermore, the inner workings of a neural network are typically difficult to understand and lack clear explanation of what each parameter does, leading the term "black box" to be used to describe neural networks by many [25]. Thus, in all practical cases, a human cannot simply select the correct parameters for a neural network as was implicitly shown for the McCulloch-Pitts Neuron in Section 1.1.1. Instead, a method known as *training* the network is employed [6].

To train a network, a *training set* is required [6, 24]. In the case of *supervised* training – which this thesis exclusively discusses – this set consists of data samples and their corresponding *targets*, or desired network outputs [6, 24]. (Depending on the context, a target is sometimes referred to as its *ground truth*. These terms are subtly different, though the distinction is unimportant with the particular data treatment in this thesis. Therefore, the term *target* will be used universally.)

To compare the quality of different sets of parameters, the *loss function* is introduced as a figure of merit or as a distance metric to measure the distance between desired output and actual output. The loss function quantifies the *loss* between a network's output and its target, where the loss is a scalar value which is larger for worse network outputs [6, 26]. For example, a common loss function is the sum square error (SSE) [27, 28], written as:

$$\ell\left(\vec{x},\vec{w}\right)=\frac{1}{2}\sum_{i=1}^{M}\sum_{j=1}^{N}\left(f_j\left(\vec{x}_i,\vec{w}\right)-t_{i,j}\right)^2 \tag{1.23}$$

where $M$ is the number of samples, $\vec{x}_i$, in the training set; $\ell\left(\vec{x}\right)$ is the total loss; N is the length of each output vector; and $\vec{t}_i$ is the target for the $i^{th}$ sample in the training set.

$f$ represents total operation of the neural network to produce the length-$N$ vector output $\boldsymbol{f}(\vec{x}_i, \vec{w})$ from the input sample $\vec{x}_i$. Rewriting in vector notation, this becomes:

$$\ell(\vec{x}, \vec{w}) = \frac{1}{2} \sum_{i=1}^{M} \left(\boldsymbol{f}(\vec{x}_i, \vec{w}) - \vec{t}_i\right)^T \left(\boldsymbol{f}(\vec{x}_i, \vec{w}) - \vec{t}_i\right) \tag{1.24}$$

where $\boldsymbol{f}(\vec{x}_i)$ and $\vec{t}_i$ are column vectors in $\mathbb{R}^N$. In this thesis, the SSE loss function is utilized unless otherwise stated. Though the loss function will necessarily be a function of parameters *and* input data, the rest of this section omits the data objects to ease notation. Additionally, to follow external sources more consistently, the parameter vector will be rewritten as $\vec{x}$.

Because worse network outputs yield larger losses, a natural goal is to minimize the loss function's output. This will indirectly improve a network's output [6]. To do this, optimization algorithms are utilized.

The general form of an optimization problem known as a *mathematical program* is [27]:

$$\text{Maximize } h(\vec{x})$$

$$\text{Subject to } \vec{x} \in \Omega \subseteq \mathbb{R}^{\mathrm{N}}$$

where $h : \mathbb{R}^{\mathrm{N}} \to \mathbb{R}$. During optimization, the elements of $\vec{x}$ are updated until an input yielding a maximum value of $h$ is found [27]. (Because the true goal is to minimize loss while training neural networks, the function $h$ represents the opposite of the loss function, i.e. $\ell = -h$.) The adjustable variables contained in $\vec{x}$ represent the neural network's parameters [6]. Each update of the parameters is known as an *iteration* [26]. It is important to note that many of these algorithms locate a function's local extremum – not its global extremum – and, furthermore, that the training of a neural network is halted once the first of several specified convergence criteria is satisfied [6, 27]. (These criteria generally include, but are not limited to: maximum iterations were performed; satisfactory error was achieved; and minimum gradient was achieved [6].) Therefore, unless training was halted due to the function having a very small gradient, these final solutions do not generally correspond exactly to a local minimum [6].

Thus, to choose parameters for a neural network to perform satisfactorily, random parameters are chosen (a process known as network *initialization*) and training is performed [6, 29]. Several training algorithms are described below.

### 1.3.1 First-Order Training

In the context of deep learning, *first-order* training methods are training algorithms based on *gradient descent*. These methods only use the network's first derivatives to minimize the loss function [30]. Because networks have multiple parameters, this is a multidimensional optimization problem, and the gradient must be calculated. Hence, in mathematical optimization, these algorithms are frequently known as gradient methods, though gradient descent is a more common term in deep learning [27]. It is important to emphasize that the gradient is taken with respect to the network parameters, *not* the input data; this is because parameters may be modified, but the input data must be treated as fixed constants.

When minimizing a function, the iteration of gradient methods takes the form [27]:

$$\vec{x}_{k+1} = \vec{x}_k - \alpha_k \nabla \ell(\vec{x}_k) \tag{1.25}$$

where $\vec{x}_k$ represents a vectorized form of a neural network's parameters after $k$ training iterations; $\alpha_k$ represents the learning rate – proportional to the distance traveled in the gradient's direction – used in the $k+1^{th}$ training iteration; $\ell$ denotes the application of the loss function on the neural network with the parameters contained in $\vec{x}$; and $\nabla$ represents the nabla operator (which provides the gradient of $\ell$) [27, 31].

With knowledge of multivariable calculus, the logic of this algorithm is easy to understand. The gradient denotes the direction of most rapid *ascent*, so moving parameters in the opposite direction – *subtraction* of $\alpha_k \nabla \ell(\vec{x}_k)$ – decreases the loss function's output. The algorithm can also be understood by manipulating the first-order Taylor expansion [27]:

$$\ell(\vec{x}_k - \alpha_k \nabla \ell(\vec{x}_k)) = \ell(\vec{x}_k) + ((\vec{x}_k - \alpha_k \nabla \ell(\vec{x}_k)) - \vec{x}_k) \cdot \nabla \ell(\vec{x}_k) + \mathcal{O}(\alpha_k^2) \tag{1.26}$$

$$= \ell\left(\vec{x}_k\right) - \alpha_k ||\nabla \ell\left(\vec{x}_k\right)||^2 + \mathcal{O}\left(\alpha_k^2\right) \tag{1.27}$$

Here, and in the rest of this thesis, $||\cdot||$ denotes the $L_2$ norm.

In practice, choosing step sizes to best minimize a function is a challenging and/or computationally-heavy task. In an attempt to solve this, different methods for choosing the step size provides the main difference between most first-order training algorithms.

The first of these methods, *steepest descent*, was introduced by Cauchy in 1847 [30]. In this method, the optimal step size is calculated during every training iteration [27, 30]. However, this requires an additional minimization process which is computationally expensive to perform during every training iteration, so other methods – such as *fixed-step-size gradient descent* – are frequently more practical [27, 30]. In this method, a small $\alpha$ is chosen and held constant throughout all training iterations [27]. Of course, "small" is relative, though the default value for fixed-step-size gradient descent in MATLAB is 0.01 and is a good first guess [32]. Unfortunately, with a fixed step size, a decrease in loss is not guaranteed [27]. If the loss function diverges during training, a smaller step size should be chosen.

Other first-order methods are common, though a number of issues persist among these. Most notably, first-order algorithms experience a slow rate of convergence, meaning that many thousands of iterations are frequently necessary to achieve satisfactory network performance [28, 33]. Despite this fact, first-order methods are popular, as they benefit from low computational complexity [28, 33].

### 1.3.2 Second-Order Training

In contrast to first-order methods, *second-order* training methods utilize a function's second derivatives to minimize the loss function with respect to a network's parameters. In the multivariable case, this information is encoded in the *Hessian matrix*, or matrix of second derivatives [28]. These second derivatives provide information regarding the curvature of the loss function's surface. This additional information enables better step size selection to be

selected. One example of second-order methods is Newton's method, whose derivation starts with a Taylor expansion [27, 28]:

$$\ell\left(\vec{x}_k + \delta x\right) \approx \ell\left(\vec{x}_k\right) + \left(\delta\vec{x}\right)^T \nabla\ell\left(\vec{x}_k\right) + \frac{1}{2}\left(\delta\vec{x}\right)^T H\left(\vec{x}_k\right)\left(\delta\vec{x}\right) \equiv q\left(\vec{x}_k + \delta\vec{x}\right) \tag{1.28}$$

Here, $H(\vec{x}_k)$ denotes the Hessian matrix calculated with respect to network parameters contained in $\vec{x}_k \in \mathbb{R}^N$, with $N$ equal to the number of parameters in the network; $\delta\vec{x}$ is a vector containing the change in each parameter as a variable; and all other objects are as described in Section 1.3.1. To continue, assume the Hessian is positive definite and say $\vec{x}_* = \vec{x}_k + \delta x_*$ denotes the coordinates which minimize the quadratic function $q(\vec{x})$ [27]. It is important to note that the function $q$ is convex due to the assumption that the Hessian is positive definite, as this allows a global minimum to be found. The first-order necessary condition requires the gradient at $q$'s minimum to be zero [27], so:

$$\nabla q\left(\vec{x}_*\right) = \nabla q\left(\vec{x}_k + \delta\vec{x}_*\right) = \vec{0} = \nabla\ell\left(\vec{x}_k\right) + H\left(\vec{x}_k\right)\left(\delta\vec{x}_*\right) \tag{1.29}$$

$$\Rightarrow \vec{0} = \left(H\left(\vec{x}_k\right)\right)^{-1}\nabla\ell\left(\vec{x}_k\right) + \delta\vec{x}_* \tag{1.30}$$

$$\Rightarrow \delta\vec{x}_* = -\left(H\left(\vec{x}_k\right)\right)^{-1}\nabla\ell\left(\vec{x}_k\right) \tag{1.31}$$

$$\Rightarrow \vec{x}_* = \vec{x}_k + \delta\vec{x}_* = \vec{x}_k - \left(H\left(\vec{x}_k\right)\right)^{-1}\nabla\ell\left(\vec{x}_k\right) \tag{1.32}$$

Thus, the global minimum of the convex function $q$ is computed in a single step. However, $q$ is merely an approximation of the loss function's local terrain, so $\vec{x}_*$ will rarely correspond to the loss function's local minimum [27]. Rather, this process is iterated, and the local minimum is approached with a quadratic rate of convergence [27, 34]. Therefore, $\vec{x}_*$ is renamed $\vec{x}_{k+1}$, and Newton's method is written:

$$\vec{x}_{k+1} = \vec{x}_k - \left(H\left(\vec{x}_k\right)\right)^{-1}\nabla\ell\left(\vec{x}_k\right) \tag{1.33}$$

As stated, this method assumes the Hessian matrix is positive definite [27]. For this to be true, good initial parameters must be selected; Newton's method only has *local convergence* [34]. However, initial parameters are typically chosen randomly, so Newton's method frequently diverges for neural networks [28]. Another downside to Newton's method is the need to calculate the Hessian, which has high computational complexity [28]. Furthermore, inverting the Hessian is costly; even the best algorithms require at least $\mathcal{O}(N^{2.373})$ operations to compute the inverse [35]. Finally, it is also possible for the Hessian matrix to be *singular* – the inverse does not exist – so that an iteration of Newton's Method cannot be performed. To address these issues, several improvements are made.

### 1.3.3 Quasi-Newton Method

The complexity of calculating the Hessian of a matrix can be mitigated under certain conditions. For example, consider utilizing the sum square error loss function described in Eqs. 1.23 and 1.24. For this section, the notation is simplified by defining [28]:

$$e_{i,j} \equiv \ell\left(\vec{w}, \vec{x}_i\right)_j - t_{i,j} \tag{1.34}$$

where each component on the right hand side is as defined in Section 1.3, and $w$ is included to represent the parameters. Then Eq. 1.23 can be rewritten as [28]:

$$\ell\left(\vec{w}, \vec{x}\right) = \frac{1}{2}\sum_{i=1}^{M}\sum_{j=1}^{N} e_{i,j}^2 \tag{1.35}$$

The derivative of this loss with respect to a single parameters becomes:

$$\frac{\partial \ell\left(\vec{w}, \vec{x}\right)}{\partial w_k} = \sum_{i=1}^{M}\sum_{j=1}^{N} e_{i,j}\frac{\partial e_{i,j}}{\partial w_k} \tag{1.36}$$

Therefore, the gradient is:

$$\nabla \ell\left(\vec{x}\right) = \sum_{i=1}^{M} \sum_{j=1}^{N} \begin{bmatrix} e_{i,j} \frac{\partial e_{i,j}}{\partial w_1} \\ \vdots \\ e_{i,j} \frac{\partial e_{i,j}}{\partial w_P} \end{bmatrix} \tag{1.37}$$

Here, $P$ is defined as the total number of parameters in the network. This can be further simplified using a matrix [28]:

$$\nabla \ell\left(\vec{x}\right) = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w_1} & \frac{\partial e_{1,1}}{\partial w_2} & \cdots & \frac{\partial e_{1,1}}{\partial w_P} \\ \frac{\partial e_{2,1}}{\partial w_1} & \frac{\partial e_{2,1}}{\partial w_2} & \cdots & \frac{\partial e_{2,1}}{\partial w_P} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial e_{M,1}}{\partial w_1} & \frac{\partial e_{M,1}}{\partial w_2} & \cdots & \frac{\partial e_{M,1}}{\partial w_P} \\ \frac{\partial e_{1,2}}{\partial w_1} & \frac{\partial e_{1,2}}{\partial w_2} & \cdots & \frac{\partial e_{1,2}}{\partial w_P} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial e_{M,N}}{\partial w_1} & \frac{\partial e_{M,N}}{\partial w_2} & \cdots & \frac{\partial e_{M,N}}{\partial w_P} \end{bmatrix}^T \begin{bmatrix} e_{1,1} \\ e_{2,1} \\ \vdots \\ e_{M,1} \\ e_{1,2} \\ \vdots \\ e_{M,N} \end{bmatrix} \tag{1.38}$$

Note: The $(M * N)$x$P$ matrix in Eq. 1.38 is the loss function's Jacobian, which is denoted $J$. Also, the column vector in Eq. 1.38 is denoted $\vec{e}$.

Taking a second derivative of Eq. 1.36, each entry of the Hessian takes the form [28]:

$$\frac{\partial^2 \ell\left(\vec{w}, \vec{x}\right)}{\partial w_k \partial w_l} = \sum_{i=1}^{M} \sum_{j=1}^{N} \left( \frac{\partial e_{i,j}}{\partial w_k} \frac{\partial e_{i,j}}{\partial w_l} + e_{i,j} \frac{\partial^2 e_{i,j}}{\partial w_k \partial w_l} \right) \tag{1.39}$$

Explicitly taking second derivatives is still required. To avoid this, assume that the second sum in Eq. 1.39 is nearly zero [28, 36], i.e.:

$$\sum_{i=1}^{M} \sum_{j=1}^{N} e_{i,j} \frac{\partial^2 e_{i,j}}{\partial w_k \partial w_l} \approx 0 \tag{1.40}$$

The Hessian can now be rewritten as:

$$H \approx \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w_1} & \frac{\partial e_{1,1}}{\partial w_2} & \dots & \frac{\partial e_{1,1}}{\partial w_P} \\ \frac{\partial e_{2,1}}{\partial w_1} & \frac{\partial e_{2,1}}{\partial w_2} & \dots & \frac{\partial e_{2,1}}{\partial w_P} \\ \vdots & \vdots & \dots & \vdots \\ \frac{\partial e_{M,1}}{\partial w_1} & \frac{\partial e_{M,1}}{\partial w_2} & \dots & \frac{\partial e_{M,1}}{\partial w_P} \\ \frac{\partial e_{1,2}}{\partial w_1} & \frac{\partial e_{1,2}}{\partial w_2} & \dots & \frac{\partial e_{1,2}}{\partial w_P} \\ \vdots & \vdots & \dots & \vdots \\ \frac{\partial e_{M,N}}{\partial w_1} & \frac{\partial e_{M,N}}{\partial w_2} & \dots & \frac{\partial e_{M,N}}{\partial w_P} \end{bmatrix}^T \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w_1} & \frac{\partial e_{1,1}}{\partial w_2} & \dots & \frac{\partial e_{1,1}}{\partial w_P} \\ \frac{\partial e_{2,1}}{\partial w_1} & \frac{\partial e_{2,1}}{\partial w_2} & \dots & \frac{\partial e_{2,1}}{\partial w_P} \\ \vdots & \vdots & \dots & \vdots \\ \frac{\partial e_{M,1}}{\partial w_1} & \frac{\partial e_{M,1}}{\partial w_2} & \dots & \frac{\partial e_{M,1}}{\partial w_P} \\ \frac{\partial e_{1,2}}{\partial w_1} & \frac{\partial e_{1,2}}{\partial w_2} & \dots & \frac{\partial e_{1,2}}{\partial w_P} \\ \vdots & \vdots & \dots & \vdots \\ \frac{\partial e_{M,N}}{\partial w_1} & \frac{\partial e_{M,N}}{\partial w_2} & \dots & \frac{\partial e_{M,N}}{\partial w_P} \end{bmatrix} = J^T J \tag{1.41}$$

Only calculating first derivatives is simpler than calculating second derivatives. Therefore, approximating the Hessian as in Eq. 1.41 greatly simplifies the computational complexity when applying second-order methods to neural networks [28]. In fact, this Hessian approximation is used to approximate Newton's Method in the *Gauss-Newton Method*, also known as the *Quasi-Newton Method*, written as [28, 36]:

$$\vec{w}_{k+1} = \vec{w}_k - \left(J^T J\right)^{-1} J^T \vec{e} \tag{1.42}$$

Due to the Hessian's approximation, this method does not experience fully-quadratic convergence, but rather superlinear convergence [37].

### 1.3.4 Levenberg-Marquardt Training

As shown in Section 1.3.3, utilizing the Gauss-Newton Method decreases the computational complexity of second-order training. However, the approximated Hessian is not guaranteed to be invertible, nor is it guaranteed to be positive definite; this is easily proven by considering a scenario where all second derivatives are equal to zero. Fortunately, all positive definite matrices are invertible [38]. Therefore, finding a method to guarantee that the approximate Hessian is a positive definite matrix is desirable, as this guarantees both properties. An

$N$x$N$ matrix $H$ is positive definite if and only if it has the property [38]:

$$\forall\ \vec{w} \in \mathbb{R}^N \setminus \{\vec{0}\}, \vec{w}^T H \vec{w} > 0 \tag{1.43}$$

Consider the approximate Hessian from Eq. 1.41. Assume it is some arbitrary, real-valued $N$x$N$ matrix $H$, and also consider the $N$x$N$ identity matrix $I$. Clearly, the identity matrix is positive definite, as:

$$\forall\ \vec{w} \in \mathbb{R}^N \setminus \{\vec{0}\}, \vec{w}^T I \vec{w} = ||\vec{w}||^2 > 0 \tag{1.44}$$

Now consider the behavior of the matrix formed by summing $H$ and $I$.

$$\vec{w}^T \left(H + I\right) \vec{w} = \vec{w}^T H \vec{w} + ||\vec{w}||^2 \tag{1.45}$$

Because $H$ is an arbitrary matrix, the value of $\vec{w}^T H \vec{w}$ may be any real number. Therefore, for a fixed $\vec{w} \neq \vec{0}$, the value in Eq. 1.45 may be any real number. Thus, a *combination coefficient*, $\mu$, is introduced and multiplied by the identity matrix to form the following [28]:

$$H + \mu I \tag{1.46}$$

$\mu$ can be chosen to guarantee that this matrix is positive definite. $\mu$ should always be chosen as a positive value. Before proving this, define the following for ease of notation:

$$S^{N-1} \equiv \{\vec{y} \in \mathbb{R}^N\ s.t.\ ||\vec{y}|| = 1\} \tag{1.47}$$

Start the proof by choosing $w_0$ with the property:

$$\vec{w}_0 \in S^{N-1}\ s.t.\ \vec{w}_0^T H \vec{w}_0 \leq \vec{w}^T H \vec{w}\ \forall\ \vec{w} \in S^{N-1} \tag{1.48}$$

Also choose:

$$\mu > -\frac{\vec{w}_0^T H \vec{w}_0}{||\vec{w}_0||^2} = -\vec{w}_0^T H \vec{w}_0 \geq -\vec{w}^T H \vec{w} \tag{1.49}$$

$$\Rightarrow \mu > -\vec{w}^T H \vec{w} \; \forall \; \vec{w} \in S^{N-1} \tag{1.50}$$

$$\Rightarrow \vec{w}^T H \vec{w} + \mu = \vec{w}^T H \vec{w} + \mu \vec{w}^T I \vec{w} = \vec{w}^T \left(H + \mu I\right) \vec{w} > 0 \; \forall \; \vec{w} \in S^{N-1} \tag{1.51}$$

Note the following:

$$\forall \; \vec{w} \in \mathbb{R}^N \setminus \{\vec{0}\}, \exists \; \vec{w}' \in S^{N-1}, c \in \mathbb{R}^+ \; s.t. \; \vec{w} = c\vec{w}' \tag{1.52}$$

$$\Rightarrow \vec{w}^T (H + \mu I) \vec{w} = c^2 \vec{w}'^T (H + \mu I) \vec{w}' \tag{1.53}$$

Of course:

$$\forall \; c \in \mathbb{R} \setminus \{0\}, c^2 > 0 \tag{1.54}$$

Thus, combining Eqs. 1.51, 1.53, and 1.54, it is clear that choosing $\mu$ as in Eq. 1.49 yields:

$$\vec{w}^T (H + \mu I) \vec{w} > 0 \; \forall \;\; \vec{w} \in \mathbb{R}^N \setminus \{\vec{0}\} \tag{1.55}$$

Therefore, for any matrix $H$, a $\mu$ can be chosen so that $H + \mu I$ is an invertible, positive definite matrix. This implies stable convergence, and equivalent proofs of this are available [27, 39].

This fact led to the development of the *Levenberg-Marquardt Algorithm* (LM), whose update rule for $\mu$ attempts to provide stable convergence while including maximum benefit from second-order information. In this method, the initial $\mu$ value is guessed and updated with each iteration to avoid calculation of the threshold value in Eq. 1.49. This provides functional $\mu$ values with very little computational complexity.

Modifying Eq. 1.42 with Eq. 1.46 yields the standard form of LM, written:

$$\vec{w}_{k+1} = \vec{w}_k - \left(J^T J + \mu I\right)^{-1} J^T \vec{e} \tag{1.56}$$

This is the standard form of LM used to train neural networks [28, 40, 41].

Like the Quasi-Newton Method, LM experiences superlinear convergence, meaning the rate of convergence should be "fast" compared to first-order methods [28]. The rate of convergence on any given iteration, however, is largely dependent on the relative values of $H$ and $\mu$ [41]. Specifically, when $\mu$ is large relative to the elements of $H$, the inverse of $H + \mu I$ resembles the inverse of $\mu I$ [41]. This implies that LM will approximate gradient descent with approximate learning rate $\frac{1}{\mu}$ when $\mu$ is large [41]. This is clear by manipulating Eq. 1.56, combining it with Eq. 1.38, and comparing the result to Eq. 1.25:

$$\vec{w}_{k+1} = \vec{w}_k - \left(J^T J + \mu I\right)^{-1} J^T \vec{e} \approx \vec{w}_k - \frac{1}{\mu} I J^T \vec{e} = \vec{w}_k - \frac{1}{\mu} \nabla f(\vec{w}) \tag{1.57}$$

Conversely, when $\mu$ is small in magnitude compared to the elements of the approximate Hessian's main diagonal, Eq. 1.56 approximates the Quasi-Newton Method (Eq. 1.42) [41]:

$$\text{Choose } |\mu| \ll |h_{i,i}| \ \forall \ h_{i,i} \in \left(J^T J\right) \tag{1.58}$$

$$\Rightarrow J^T J + \mu I \approx J^T J \tag{1.59}$$

$$\Rightarrow \left(J^T J + \mu I\right)^{-1} \approx \left(J^T J\right)^{-1} \tag{1.60}$$

$$\Rightarrow \vec{w}_{k+1} = \vec{w}_k - \left(J^T J + \mu I\right)^{-1} J^T \vec{e} \approx \vec{w}_k - (J^T J)^{-1} J^T \vec{e} \tag{1.61}$$

Combining the above, it is clear that $\mu$ must be chosen carefully [28, 41]. The general method is to choose a "large" initial value for $\mu$ [28, 41]. ("Large" is, of course, a relative term, and it represents a blind guess about the magnitudes of the loss function's second

derivatives. For reference, the default initial value of $\mu$ is 0.001 in MATLAB [40].) By starting with a large value of $\mu$, gradient descent will move the parameters towards a good initial "guess" of the optimal parameters [28, 39, 40, 41].

This progressively-better guess will eventually yield a positive-definite matrix allow, at which point $\mu$ should be minimized so that the superior convergence rate of the Quasi-Newton Method is realized [28, 39, 40, 41]. In practice, $\mu$ is typically adjusted at the end of each iteration [28, 39, 40, 41]. If error increases, the parameters are returned to their prior value and $\mu$ is increased [28, 39, 40, 41]. Conversely, if an iteration is successful, the value of $\mu$ is decreased [28, 39, 40, 41]. (Some sources recommend multiplying $\mu$ by 10 after unsuccessful iteration attempts and dividing by 10 after successful iterations [28]. These are the default adjustments in MATLAB, as well [40].)

In some suggested implementations of LM, an iteration is only attemped a certain number of times before the parameters are adjusted despite an increase in loss [28]. This implementation is used throughout this thesis, as seen in the code in Chapter 4.

Other modifications to LM exist, as well. Notably, the computational complexity of the method was dramatically decreased by including accelerated matrix inversion [42, 43, 44, 45]. This method sacrifices some accuracy in the matrix inversion in exchange for significantly faster computation[42]. This allows iterations to be performed rapidly, decreasing the training time for networks using LM [42].

The Levenberg-Marquardt Algorithm provides a method to achieve superlinear convergence with a relatively simple calculation of the Hessian [28, 40, 41]. However, for large networks, the computational cost of inverting the approximate Hessian causes LM to underperform when compared to gradient methods [35, 40]. Additionally, memory issues can become an issue as the matrix being inverted increases in size at a rate of the number of parameters squared [40, 41]. This served as motivation for Youssef et al., who introduced Multi-Stage Training to decrease size of the Hessian matrix being inverted and allow Levenberg-Marquardt training of large networks [35].

## 1.4 Multi-Stage Training and Multi-Stage Networks

To summarize the discussion of training algorithms so far, recall that first-order training algorithms require minimal computational complexity, but linear convergence implies that many iterations are required to locate a solution (Section 1.3.1). Conversely, second-order training algorithms can converge to a solution quadratically; therefore, these methods typically converge to a solution in fewer iterations than first-order algorithms (Section 1.3.2). However, second-order methods suffer from unstable convergence [28]. This issue was solved with the introduction of the Levenberg-Marquardt algorithm; this algorithm yields stable convergence and provides a superlinear rate of convergence that approaches quadratic convergence after successive successful iterations that result in a very small value of the combination coefficient $\mu$ (Section 1.3.4). However, large networks with many parameters yield very large matrices that are memory-intensive and require significant resources to invert, meaning Levenberg-Marquardt is not frequently used in practice (Section 1.3.4).

One solution to this problem is Multi-Stage Training (MST), which enables second-order training of large networks with limited compuational resources [35]. In its simplest form, an MST network – known as a Multi-Stage Network (MSN) – is analogous to a small FCN. As described in Section 1.2.1, the first layer of an FCN – known as a *stage* in an MSN – interacts directly with input data while subsequent layers use the prior layer's output as their input. The final layer's output is the network's output, and this may be used for line fitting, classification, or other applications for which other deep neural network are used. Understanding the MSN architecture will better enable understanding of MST, so this is discussed first.

The difference between an MSN and a standard FCN is that each "neuron" is itself a small FCN. These intermediate FCN "neurons" are called *Multi-Layer Perceptrons* (MLP). In classification tasks, each MLP is trained to output a 1 for one particular class and a 0 for all other classes [35]. In other words, each MLP is trained to learn to distinguish a single class from among a larger group. Thus, it is natural to choose an integer multiple of the number of classes as the number of MLPs in any given stage. This way, an equal number of

MLPs attempt to identify each class. This architecture is shown attempting to classify an orthogonal frequency-division multiplexing (OFDM) packet in Fig. 1.6.

While the MLP structure is flexible, most implementations use an MLP with two hidden layers, each with 10-15 neurons [35]. Each hidden layer's activation functions depend on the particular application, though the output layer generally has a linear transfer function [35]. A notable exception to this general structure is discussed in Section 3.3.1.

Due to their relatively small sizes, each MLP may be efficiently and quickly trained using second-order methods by avoiding the computation and memory bottlenecks generally seen with larger networks. These superior training algorithms result in good solutions for each MLP while performing relatively few training iterations. It is important to note that each MLP is independently trained and initialized [35]. A notable benefit provided by this is that different MLPs will generally learn different network parameters corresponding to different local minima of the loss function. In other words, each MLP generally learns unique features of the training data. Thus, after data is processed by a single stage of the MSN, multiple features of that input have been targeted by specialized networks.

Do note, however, that an MLP's output is not – and should not generally be – exactly a 0 or 1. Because an MLP's output lies on a continuum, additional information is retained prior to passing data to the next layer that would otherwise be removed by limiting the output to boolean values.

For example, consider an identification task. When using MST, each MLP is trained to identify a single object by outputting a 1 for a single class and a 0 for anything else. Perhaps two classes, A and B, share a feature that other objects do not share. Now consider an MLP which attempts to identify class A using this feature. If the feature is distinguishable between classes A and B, the MLP may achieve 100% accuracy identifying class A, i.e. the MLP's output is greater than or equal to 0.5 whenever a pattern from class A is inserted, but the output is less than 0.5 whenever any other class is inserted. However, because the

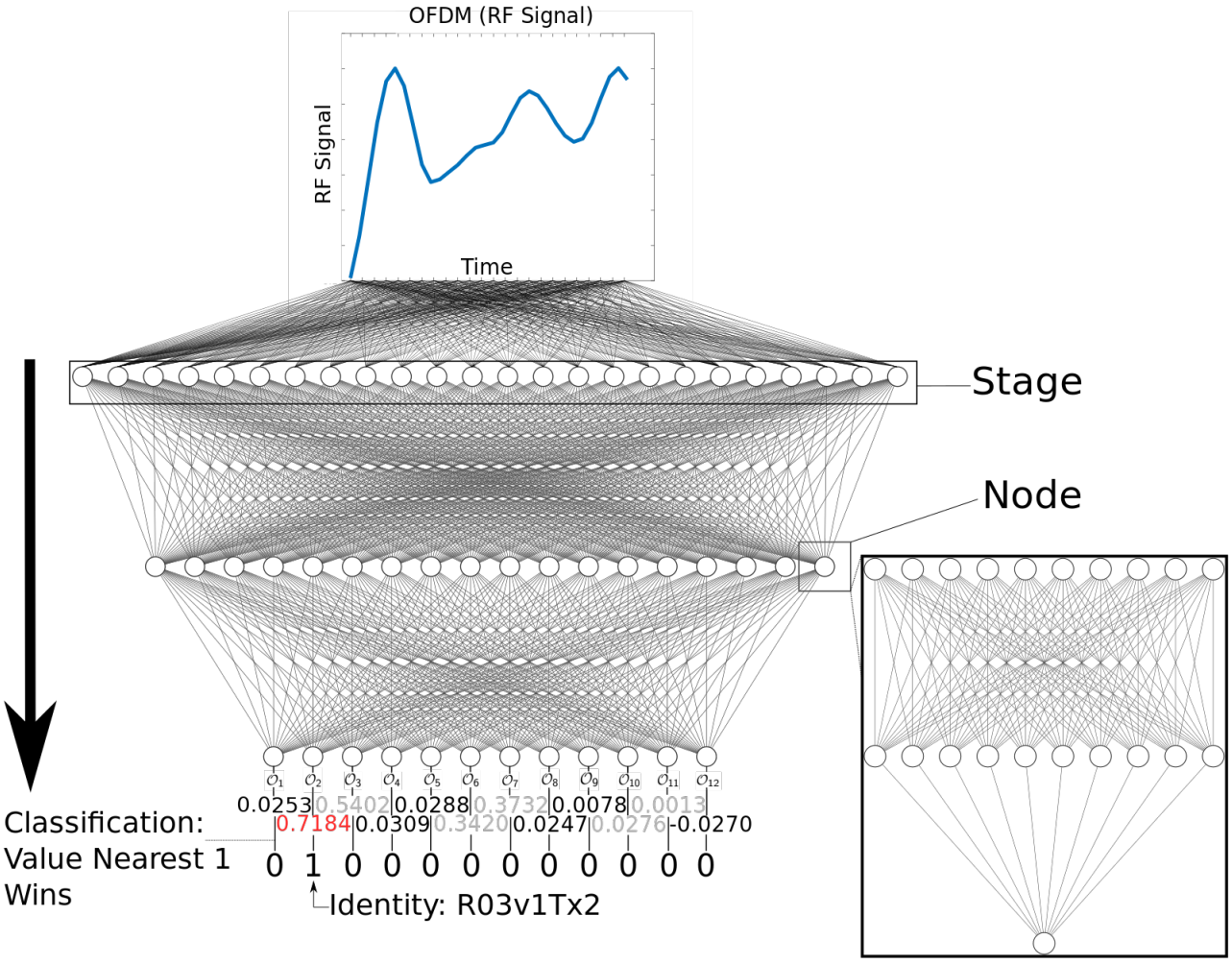


**Figure 1.6:** A sample MSN is shown attempting to identify an RF transmitter based on a section of an OFDM packet. Each data point in the sample data is fed into every node. Each node is an MLP, as shown. As FCNs, these all data entering each node passes independently through each neuron in the each layer. Each MLP outputs one value for the input sample. In this example, there are 12 output nodes for 12 RF transmitter classes, with each output node attempting to identify one transmitter. The value nearest 1 is selected as the 'winner,' so the output is changed into a vector of boolean values with a 1 at the position corresponding to the transmitter identified. This figure was produced in part using [1].

feature analyzed by this MLP is also seen in B, perhaps the MLP's mean output is:

$$< \text{Output} >= \begin{cases} 0.72 & \text{if Class A} \\ 0.39 & \text{if Class B} \\ \text{-}0.04 & \text{Otherwise} \end{cases} \tag{1.62}$$

In this situation, this MLP identifies Class A while also identifying the presence of the analyzed feature in Class B. This information will be useful when attempting to identify Class B in further stages.

It should be clear that there are many hyperparameters to consider when designing an MSN. In the simple form described thus far, these are: The number of stages; the number of MLPs per stage; the number of hidden layers per MLP per stage; and the activation functions for each MLP's layers. In Chapter 2, further modifications are explored, representing recent research.

Considering the MSN structure and its properties discussed above, it should be clear that MST varies significantly from the standard approach of training all network parameters simultaneously. Now, the training process is discussed more clearly.

Data processing can be performed in any number of ways, though each sample should be vectorized before inserting it to the MSN. This input is placed into each MLP in the first stage in the same way. In other words, the network input is used as the input to each individual MLP in the first stage. However, to aid generalization of the network and avoid memorizing data while also reducing computation time during training, each MLP is trained with a small batch from the total training set [35]. Thanks to a different subset of training data, the landscape of the loss function will appear slightly different at each node. This promotes different final parameters which corresponds to analyzing different features within the data.

Once the first stage is trained, the stage's output must be calculated and passed to the second stage, where it is used as the updated training data. To perform a forward pass

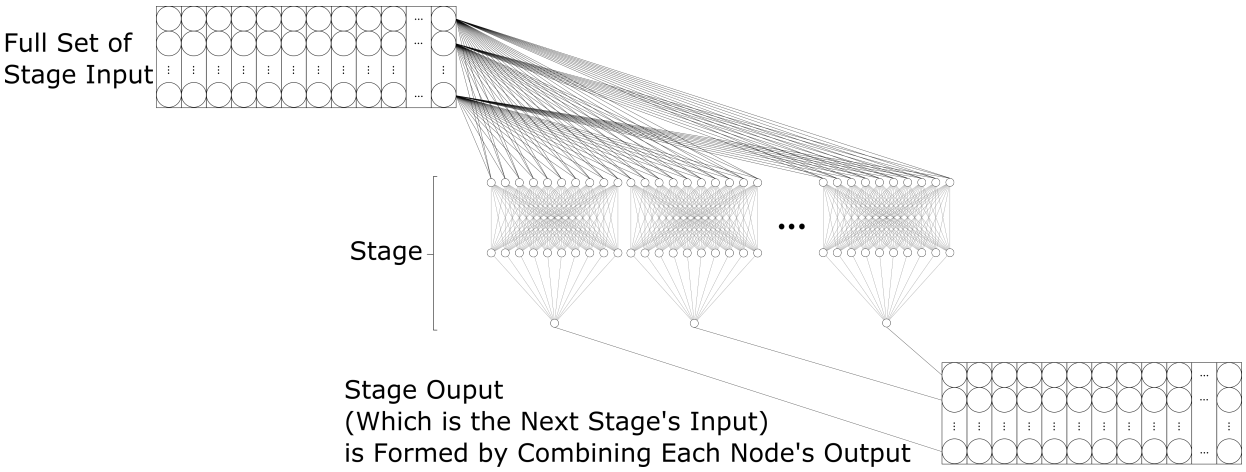


**Figure 1.7:** One stage of a Multi-Stage Network is shown performing a forward pass on a full set of input data. Each column in the stage's input is one input sample. In the first stage, several MLPs are shown side-by-side, each acting on all of the data. Their outputs of size one are then concatenated into one-dimensional vectors. There is one output vector (stage output) per one input vector (input sample). One stage's output is the next stage's input, except in the case of the final stage. The final stage's output is the network's output. This figure was produced in part using [1].

through the first stage, insert the entire training set – not just the subset used for training – through each of the MLPs in this stage. The output of each MLP is, of course, of size 1 for each data sample. The output's length for any given data sample, however, will be equal to the number of MLPs in the stage. To achieve this, the output from each MLP is concatenated. This process is illustrated in Fig. 1.7.

As training continues through each stage, this identical process is repeated: Train the $n^{th}$ stage with the output from the $(n\text{-}1)^{th}$ stage, randomly producing subsets of the stage's training data for each MLP; once each MLP is trained, pass all of the training data forward through each MLP and concatenate the outputs. This is shown in Fig. 1.8.

Additional data processing at each stage should create better predictions than the prior stage. As such, the number of MLPs can be decreased from one stage to the next [35]. Therefore, the size of each input is decreased for succesive stages as the data's information is compressed. Therefore, the number of parameters in each MLP in later stages will generally decrease, barring significant changes in MLP architecture between stages. This makes it computationally feasible to increase the number of data samples used to train each MLP in

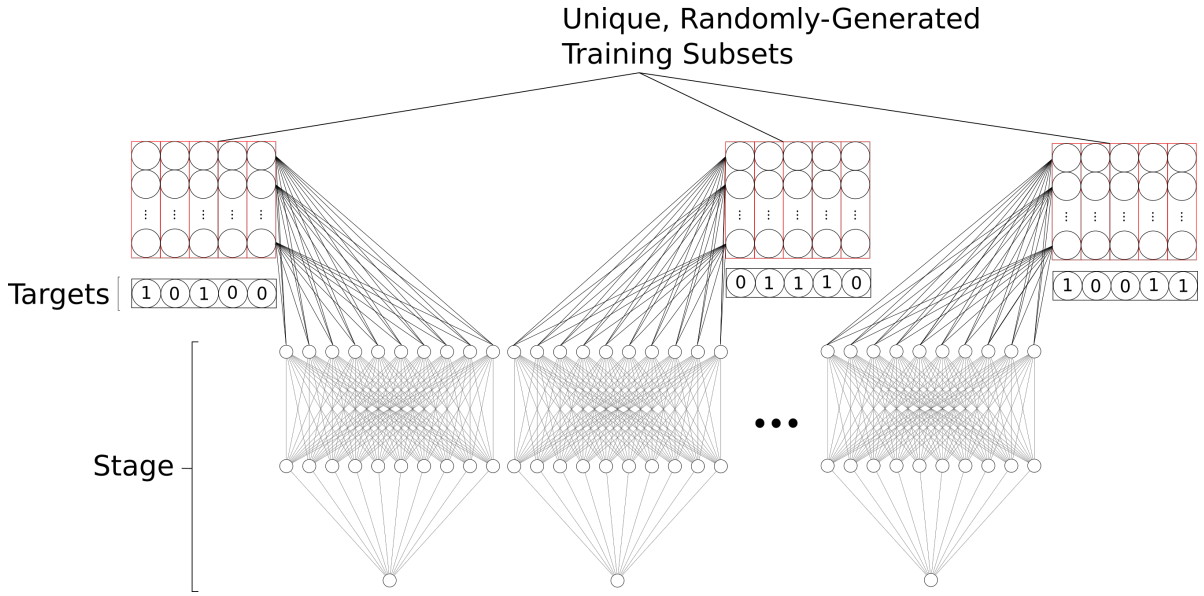


**Figure 1.8:** Here, the training process for one stage of MST is shown. First, subsets of the full training set are produced for each of the nodes in a stage. The subset at each node is unique and random. Each MLP attempts to identify a single class, so the network target for each is a 0 or 1 depending on the training sample's identity. This figure was produced in part using [1].

later stages [35]. See Chapter 3 for an example of this.

Once the MSN is trained, it can, of course, be used to classify data in a validation set. To do this, the data is operated on each stage sequentially as described above. The final stage's, is a vector of length equal to the number of MLPs in this stage – not a single number. To choose a single-valued output at this stage corresponding to a single class, multiple techniques can be devised. In this thesis, the output of all MLPs which seek to identify a given class are averaged. This results in one output per class. The class whose output is closest to 1 is chosen as the winner. Code for this can be seen in Chapter 4.

In practice, multi-stage training was shown to improve classification accuracy of different radio frequency (RF) transmitter and receiver combinations [35]. In fact, the MSN trained with Levenberg-Marquardt was able to classify these devices with an accuracy of 96.8% while traditional, large FCNs and CNNs – each trained with first order methods – yielded accuracies of 84.8% and 67.3% accuracy, respectively.

A major development in this thesis is utilizing this general outline of MST while expand-

ing upon the permitted architectures for MSN. This ultimately reveals that imporvements can be made to MSN which provide equal or better accuracy with significantly decreased compuation time.

# CHAPTER 2

# From Multi-Stage Networks to Convolutional Multi-Stage Networks

Section 1.4 introduced Multi-Stage Training (MST) and the original outline for designing multi-stage networks (MSN). These are large networks constructed using many smaller networks, now renamed *nodes*. Training is performed independently on each node in the overall network. Thus, only a subset of the MSN's parameters are trained with any given iteration. This distributed training allows the highly-effective Levenberg-Marquardt (LM) training algorithm to be used to train large networks without suffering the effects of memory and computational limitations generally encountered with large networks and second-order methods.

However, the MSN structure proposed in Section 1.4 is highly rigid, allowing only multilayer perceptrons (MLP) to act as the network's nodes. Fundamentally, this restriction is not required. As such, Youssef et al., introduced the *Convolutional Multi-Stage Network* (CMSN) [46]. A CMSN is similar in structure to the MSN, but the nodes in the first stage are convolutional neural networks (CNN) rather than MLPs. This allows the CMSN to benefit from the CNN properties discussed in Section 1.2.2, most notably robustness to shifting of the input data and a reduced number of network parameters in each node of the first stage. In theory, the first quality should enable more robust analysis of data which may not be uniformly oriented, while the second quality simplifies the overall network when input data is exceptionally large.

As a proof-of-concept for this network structure, a radar target classification experiment was performed [46].

## 2.1 Problem Description and Motivation

Radar is a valuable technology for visualizing objects at a distance. In large part, this is due to its ability to perform in a wide range of atmospheric and weather conditions. Radar's lack of attentuation in the face of adverse conditions is due its use of radio frequency (RF) radiation, which has long wavelengths.

At a basic level, radar systems operate in a fairly simple manner. First, a strong pulse of RF radiation is emitted. This travels through the air until an object is encountered. These objects may absorb, reflect, or scatter the RF pulse, and the total interaction is generally a combination of these. The reflected RF radiation is received by the radar system and used to locate an object. The relative amount of energy reflected depends on the RF pulse's wavelength and polarization, as well as the interacting object's size, shape, and material composition [46].

This object-dependent interaction results in unique returning signals. Theoretically, this object-by-object uniqueness can be used to identify the object that returned the RF pulse: Planes, helicopters, missiles, etc. A very effective radar system and data analysis scheme should be able to distinguish between different subsets of each of these classes, e.g. accurately distinguishing between an F/A-18 Hornet fighter jet and a civilian Gulfstream IV.

This information has many valuable applications, and perhaps the most important is avoiding tragic loss of life in military accidents. Consider, for example, the 2020 missile strike on Ukranian International Airlines flight PS752 over Iran which tragically resulted in 176 deaths [47]. The official cause of this accident was "missile fire due to human error," and it is concievable that having a radar system which automatically identified the aircraft as civilian may have prevented this loss of life.

This problem and others motivate the need for automatic target recognition (ATR) using radar data. This field has been developed over several decades. The prototype dataset, the Moving and Stationary Target Acquisition and Recognition (MSTAR) dataset, was collected by the US Air Force in 1995 and 1996 [48]. This dataset consists of synthetic-aperture radar

(SAR) readings of multiple ground-based military machines [48]. Today, clever techniques consistently enable greater than 99% accuracy on these datasets [49, 50, 51, 52, 53, 54, 55, 56].

However, the MSTAR dataset is relatively small, and many techniques are specifically tailored to solving that dataset – not providing a general method that scales with a large number of radar targets and that can be applied to many data types. These techniques involve a variety of data pre-processing techniques that are tailored to SAR, and neural networks which are the proper size for specifically identifying the number of classes in the MSTAR dataset.

Fortunately, the composite structure of CMSN makes it inherently scalable. Therefore, a CMSN should be capable of learning many classes by simply adding more nodes to the network, and the distributed training method allows the potentially massive network to be trained without additional computational resources. Additionally, the inclusion of CNNs in a CMSN should allow the network to automatically learn a range of data features that would be missed by MLP nodes. This potentially removes the need for precise data preprocessing techniques, enabling its automated application to many types of radar signals. Thus, CMSN potentially provides an opportunity for a scalable end-to-end radar classification scheme.

All further work in this chapter may be attributed to [46].

## 2.2 Data Collection

In collaboration with the Prof. Yahya Rahmat-Samii laboratory, we used an anechoic chamber to collect radar data from 17 objects taken from electrical engineering and chemistry laboratories. Each object was oriented in three different ways, defined as 0, 45, and 90°. The objects and a description of the orientations are shown in Fig. 2.1. Data collection was primarily performed by Greg Schuette.

For each object-orientation combination, 12 trials were performed, and the objects were intentionally moved by small amounts between trials to add noise to the data. Each trial was represented by an S21 parameter trace consisting of 1601 complex-valued datapoints,

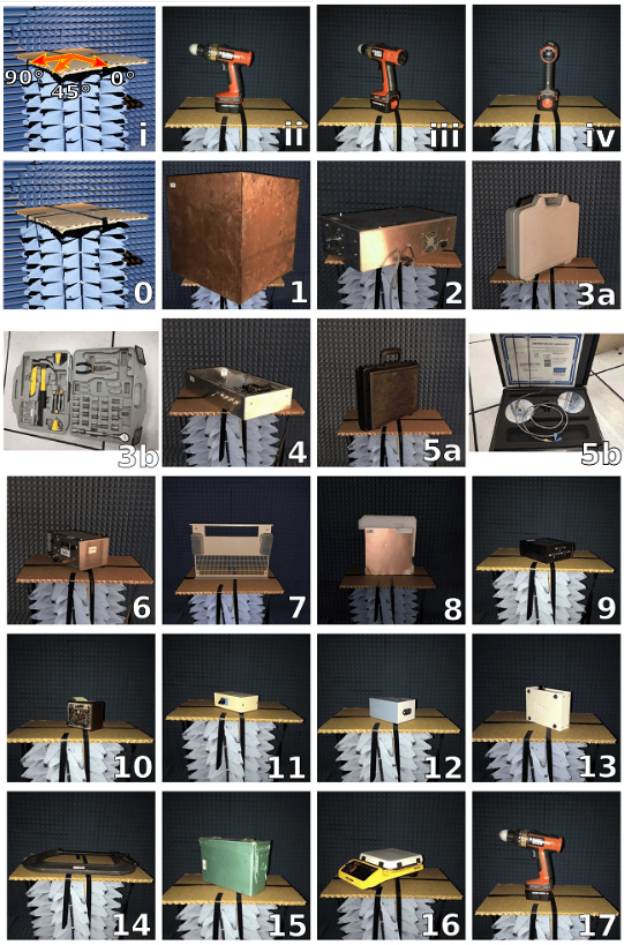


**Figure 2.1:** The 17 radar targets are shown. Object size can be approximated using the 47 cm wide by 30.5 cm deep cardbord platform. Definitions for the 0, 45, and 90° orientations are shown in (i) via red arrows on the empty cardboard platform. Object #17 is shown in its 0, 45, and 90◦ orientations in (ii), (iii), and (iv), respectively. Unless otherwise noted, photos are taken from the radar's perspective at a 45° orientation. The objects vary in size and composition, affecting rRCS. (0) Platform: Roughly 60° from the radar's perspective. (1) Object 1: Empty metal (copper) box with hole on top face. (2) Object 2: Metal box (closed cover) with home-built circuit. (3a) Object 3: Plastic toolbox (closed). (3b) Object 3: Open; data was collected with the toolbox closed. (4) Object 4: Metal box (open cover) with custom circuit. (5a) Object 5: Plastic box (closed). (5b) Object 5: Open; data was collected with the box closed. (6) Object 6: DC power supply (metal cover). (7) Object 7: Front cover of a power amplifier (metal) at 90° orientation. (8) Object 8: Rogers duroid laminate (copper) at 90° orientation. (9) Object 9: Data transfer switch box (with plastic cover). (10) Object 10: Variable capacitor box (metal cover). (11) Object 11: Data transfer switch box (with metal cover). (12) Object 12: Port converter (metallic). (13) Object 13: Data transfer switch box (with plastic and metal cover). (14) Object 14: Vise (metal) at 0° orientation. (15) Object 15: Metal box. (16) Object 16: Chemistry hotplate stirrer. (17) Object 17: Black Decker drill at 0° orientation. This figure was produced by Greg Schuette.

each representing linear amplitude versus frequency. The 1601 points were equally spaced on the range of 675 MHz to 8.5 GHz; this represents an ultra-wide bandwidth (UWB) range.

The anechoic chamber has band rejection over frequencies from 1 MHz to 10 GHz, thus shielding environmental noise and chamber reflections over the entire range of frequencies at which data was collected. The data was collected using two vertically-oriented TSA900 900 MHz - 12 GHz PCB Vivaldi Antennas (RFSPACE Inc., Atlanta, GA). These were attached to a model E5071C 9 kHz-8.5 GHz ENA Series (Agilent, Santa Clara, CA) vector network analyzer (VNA). The data was collected in the VNA's Smith Chart mode. This equipment was calibrated by Yubin Cai.

Multiple traces of the empty chamber were collected to calculate the background signal and validate the experimental setup, shown in Fig. 2.2. A plot depicting a representative trial of the background trace is shown in Fig. 2.3. The background signal stems from the coupling of the antennae, as confirmed by an S11 parameter simulation in free space for our setup. This simulation was performed by Yubin Cai, and the S11 parameter simulation was converted to the S21 parameter, also shown in Fig. 2.2. The S11 parameter was converted to the S21 parameter using Eq. 2.1 [57, 58].

$$|\mathrm{S21}| \equiv \frac{|V_{\mathrm{Rx}}|}{|V_{\mathrm{Tx}}|} = (1 - |\mathrm{S11}|^2)(\frac{\lambda}{4\pi r})\mathrm{G_A}, \tag{2.1}$$

For Eq. 2.1, the antennae are assumed to have identical gain and reflection coefficients without loss of polarization. $\lambda$ represents wavelength of the radio-frequency radiation. $r$ is the distance between the receiver (Rx) and transmitter (Tx) antennae. $\mathrm{G_A}$ is the $\lambda$-dependent antenna gain. Finally, $|V_{\mathrm{Rx}}|$ ($|V_{\mathrm{Tx}}|$) represents the voltage at the receiver (transmitter) antenna.

## 2.3 Data Analysis: Non-AI

As discussed in Section 2.1, many effective methods exist to accomplish ATR on the MSTAR dataset. However, these typically combine semi-manual data pre-processing techniques and

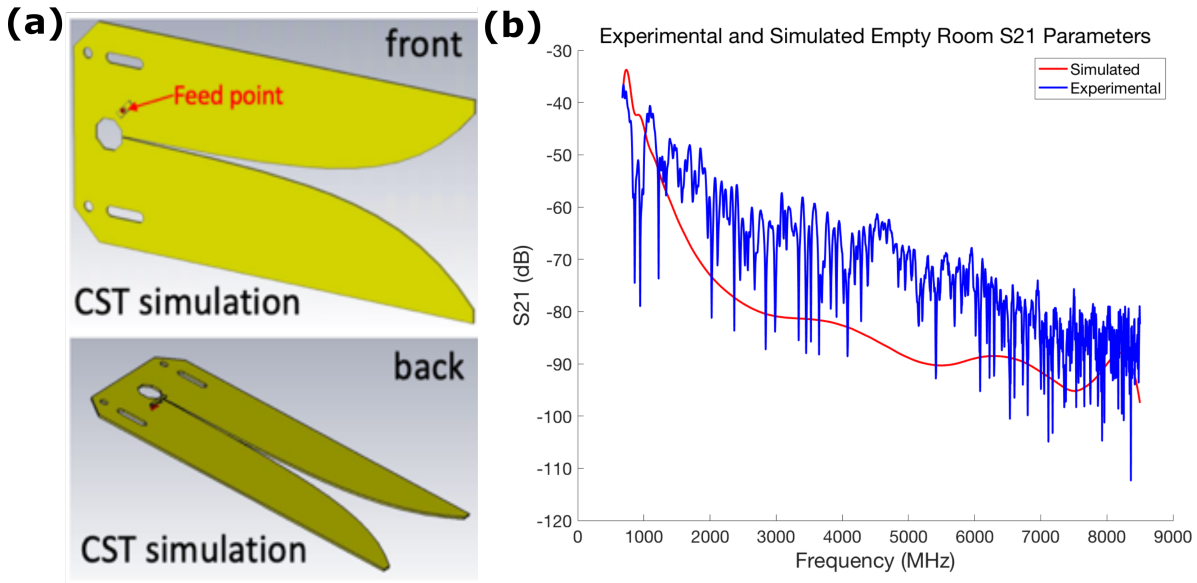


**Figure 2.2:** (a) A 3D model of the Vivaldi antennas used in the simulation of the S11 parameter is shown in two orientations. This was produced by Yubin Cai. (b) The S21 parameter of the empty chamber is shown. The red line shows the S21 parameter trace as calculated via the simulation and Eq. 2.1. The blue line shows the S21 parameter trace a trace collected with the experimental setup. The values are shown in decibels, as is traditional. This plot was produced by Greg Schuette and Yubin Cai.

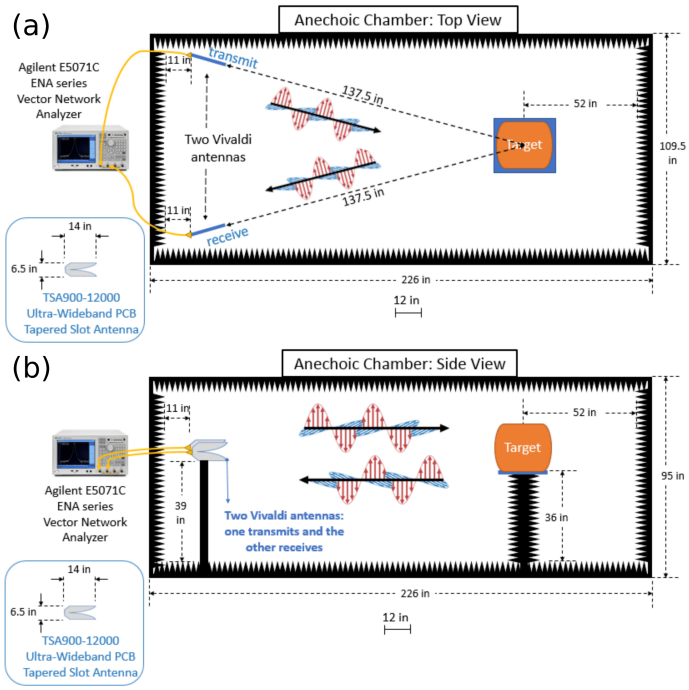


**Figure 2.3:** The setup of the chamber is depicted with dimensions shown. Figure created by Yubin Cai.

artificial intelligence techniques [49, 50, 51, 52, 53, 54, 55, 56]. Here, two separate manual data analysis techniques are utilized, each emphasizing the difficulty of manually classifying objects based on the S21 parameter.

The work in this section was performed primarily by Greg Schuette.

### 2.3.1 Naive Data Analysis

We first attempted to classify objects by analyzing the log-magnitude plot of the S21 parameter for each object-orientation combination. Because this corresponds to the magnitude of the received signal, higher values correspond to a larger amount of reflected energy. Therefore, objects with large radar cross sections (RCS), which reflect relatively large amounts of energy and yield relatively large signals, should show high values in these plots at relevant frequencies. These radar cross sections indicate how much energy may be reflected by an object and is due to a combination of object size, orientation, and material composition. See Table 2.1 for the calculated relative RCS (rRCS) of each object; details of the calculation are discussed in the figure description. In addition to the overall RCS, objects with different dimensions or material compositions will reflect certain frequencies more effectively than others, providing another parameter to use for identification. Table 2.1 also shows signal-to-noise ratio (SNR) results for each object-orientation combination.

To visualize these object-dependent qualities for classification attempts, the log-magnitude plots of the S21 parameter for each of the 17 objects in their different orientations were created. Plots of each object in their respective 0° orientations are shown in Fig. 2.4.

Several objects could be identified using this method. The identifiable objects were those with large RCSs, which tended to be large, metallic objects. This is shown in Fig. 2.4 (a). Here, different objects' signals maintain different overall magnitudes and often include several unique points with unusually effective reflection.

Despite these object-dependent qualities and several successes, manual identification of most targets via this method proved difficult. This is visualized in Fig. 2.4 (b). Here, all

| Object | SNR, 0° | SNR, 45° (dB) | SNR, 90° (dB) | rRCS, 0° | rRCS, 45° | rRCS, 90° |
|---|---|---|---|---|---|---|
| 1 | 20(10) | 13(8) | 19(9) | 0.9(3) | 0.10(2) | 1.0(4) |
| 2 | 14(7) | 14(7) | 8(7) | 0.34(5) | 0.10(2) | 0.4(1) |
| 3 | 13(7) | 14(8) | 12(7) | 0.10(2) | 0.10(2) | 0.14(2) |
| 4 | 12(7) | 14(7) | 11(6) | 0.22(6) | 0.09(3) | 0.17(3) |
| 5 | 14(8) | 13(7) | 14(8) | 0.10(2) | 0.10(2) | 0.17(4) |
| 6 | 12(8) | 13(7) | 10(6) | 0.23(3) | 0.10(3) | 0.17(2) |
| 7 | 13(8) | 13(8) | 14(9) | 0.10(3) | 0.10(2) | 0.5(2) |
| 8 | 12(7) | 14(8) | 13(8) | 0.10(3) | 0.10(3) | 0.20(3) |
| 9 | 12(6) | 14(7) | 10(6) | 0.10(3) | 0.10(3) | 0.11(3) |
| 10 | 10(6) | 13(7) | 12(7) | 0.12(3) | 0.09(3) | 0.09(3) |
| 11 | 10(6) | 14(8) | 11(6) | 0.12(2) | 0.09(3) | 0.10(3) |
| 12 | 10(6) | 13(7) | 10(6) | 0.14(3) | 0.09(3) | 0.11(3) |
| 13 | 11(7) | 12(7) | 10(6) | 0.23(2) | 0.10(3) | 0.11(3) |
| 14 | 11(6) | 13(7) | 12(7) | 0.11(3) | 0.09(3) | 0.10(3) |
| 15 | 14(8) | 11(6) | 11(7) | 0.44(5) | 0.10(3) | 0.14(2) |
| 16 | 11(7) | 13(7) | 11(6) | 0.10(3) | 0.10(3) | 0.10(3) |
| 17 | 12(7) | 13(7) | 13(7) | 0.10(3) | 0.10(3) | 0.10(3) |

**Table 2.1:** SNR and rRCS values for each object (1–17) and their three orientations are shown. The standard deviation of each signal is shown in parentheses. Twelve traces were recorded for each object in each orientation. 112 traces of the empty anechoic chamber were recorded. The arbitrary SNR values were calculated by first calculating the magnitude at each point in each trace. An average of the 112 empty chamber traces was produced to minimize noise, and this mean trace was subtracted from each object-orientation trace to remove the background signal. Next, averages of 100 consecutive points were calculated for each possible position in each trace to reduce noise when calculating the maximum signal. The highest average was taken as the signal's magnitude. A flat region in each trace was located, and the standard deviation of 100 points in the flat region was calculated to determine noise. The maximum signal was divided by this noise to calculate SNR, which was then converted to decibels. Relative radar cross sections (rRCS) were determined by first calculating the area under the background-subtracted S21 traces. This was performed in MATLAB using the trapezoidal rule. This represents a total radar signal or "pulse amplitude." The pulse amplitudes were averaged for each object-orientation combination. Each average was then divided by the largest signal to yield a value between 0 and 1. All calculations were performed by Greg Schuette.

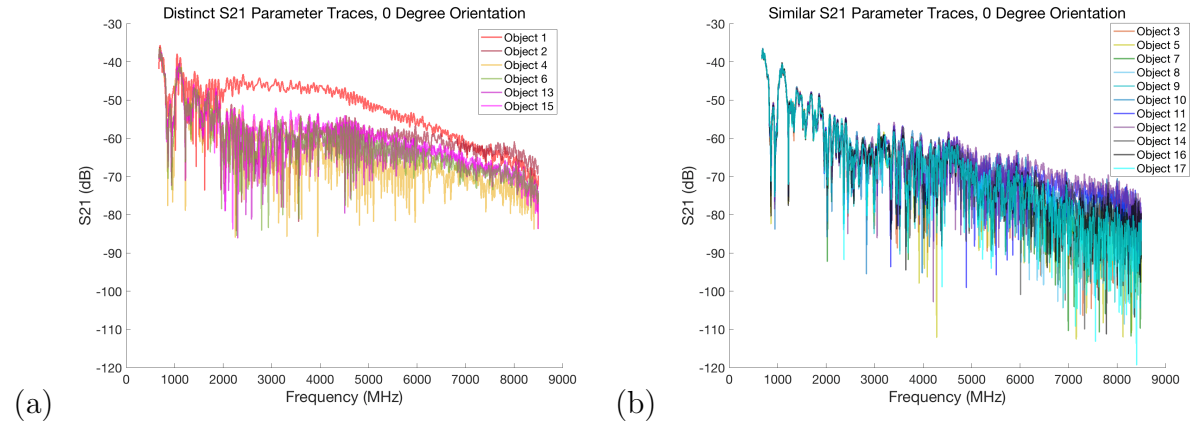


**Figure 2.4:** S21 parameters (raw data, log-magnitude traces) for all 17 objects at their respective 0° orientation. Traces show the raw data as acquired without any post processing or averaging. (a) Several of the targets (objects #1, 2, 4, 6, 13 and 15) have S21 parameters with obvious differences in the log-magnitude plot. (b) Other targets (objects #3, 5, 7, 8, 9, 10, 11, 12, 14, 16 and 17) exhibit S21 parameters with no obvious visual differences.

signals appear nearly identical to the background signal in the low frequency range due to the the strength of antenna coupling at low frequencies. Noise between trials made it impossible to find identifying features for objects with low RCSs in the high frequency range.

Furthermore, several objects' orientations are distinguishable after object classification. These tend to be large, metallic objects with low symmetry. These objects have different physical cross sections when reoriented, yielding different RCSs, as well. One partial exception to this trend is Object #1, whose cross sections are very similar in the 0 and 90° orientations. Object #1's S21 traces at different orientations are shown in Fig. 2.5 (a). Here, the 0 and 90° orientations yield very strong signals. This is likely because Object #1 is a large copper box, and one of two faces on the box is directly opposite the antennae in the 0 and 90° orientations. The fact that these two faces of equal size, relative positioning, and composition are still distinguishable is likely due to the crimped copper seam, visible in Fig. 2.1, which is on the right and left in the 0 and 90° orientations, respectively. This provides slight asymmetry between these orientations. Note also that when the box is in the 45° orientation, the signal is very weak. This is most likely because the box's faces are oriented in a way that scatters light rather than reflecting it towards the antennae.

However, most objects lack any clear distinguishing characteristics between orientations.

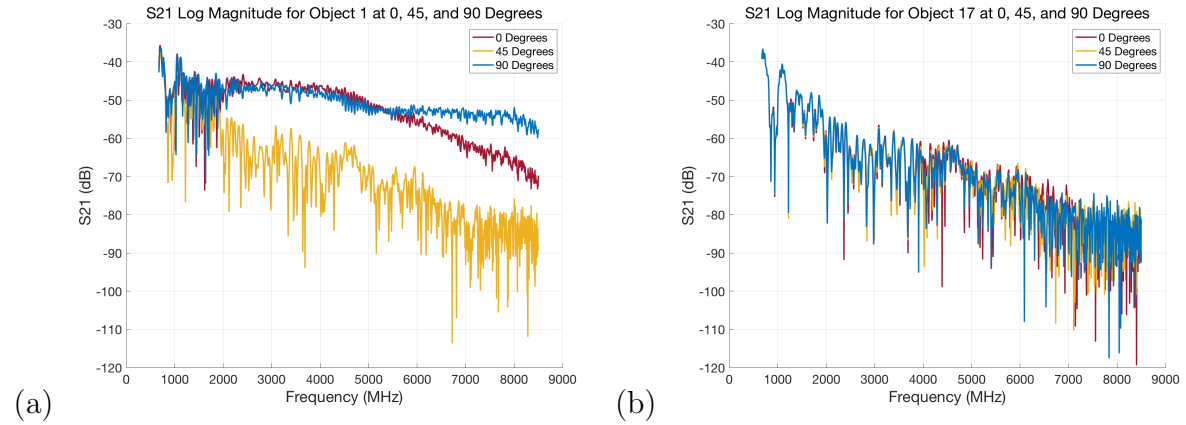


**Figure 2.5:** S21 parameters for two targets (objects #1 and #17) at three different orientations (the objects and a definition of orientation are indicated in Fig. 2.1). No noise removal or averaging was performed on the traces shown. (a) Object #1 is shown at 0, 45, and 90° orientations, each of which is easily identifiable. Objects #2, 4, 6, and 13 (not shown here) also demonstrate easily distinguishable orientations. (b) Object #17 is shown at 0, 45, and 90° orientations, each of which are indistinguishable. Orientations for Objects #9, 10, 11, 14, and 16 (not shown here) were also visually indistinguishable from their log-magnitude plots. Figure produced by Greg Schuette

One example, Object #17, is shown in Fig. 2.5 (b). This is due to weak overall signals and relatively high noise in the high-frequency range between trials, making readings in these regions inconsistent between trials.

Further development of this method may work for identifying particular objects in a particular situation. However, the tested conditions are ideal: An anechoic chamber removed environmental noise; very precise equipment was used; the objects were at a short range; and only one object was in the field of view for any given measurement. Thus, this method is unrealistic in real world applications, so a more robust manual analysis was investigated.

### 2.3.2 Time-Frequency Analysis

The analysis in Section 2.3.1 relied purely on the frequency domain of the S21 parameter. Theoretically, one could perform a Fourier transform to analyze the data in the time domain. However, a more effective method for human interpretation is time-frequency analysis, allowing both domains to be visualized simultaneously.

This analysis was based on the theory from [59]. Note that the x- any y-axes represent the time and frequency, respectively, in time-frequency plots. Thus, vertical lines in these plots correspond to many frequencies reaching the receiving antenna simultaneously, which indicates a broad reflection [59]. Conversely, horizontal lines correspond to a single frequency (or small range of frequencies) reaching the receiving antenna over an extended period of time; this implies that these frequencies are resonant within the target object, becoming trapped and being emitted towards the receiving antenna over time [59]. Curves and diagonal lines also appear: Signals starting at low frequency/early time and proceeding to high frequency/late time correspond to material dispersion; signals starting at high frequency/early time and proceeding to low frequency/late time correspond to structural dispersion [59]. These features indicating specific interactions within each object are thus more easily visualized with time-frequency analysis.

The time-frequency method used was the two-dimensional continuous wavelet transform (CWT). These use inner products to compare a signal to an analyzing function, known as the *mother wavelet* [35, 60]. The general form of this computation is [60]:

$$C\left(a,b,f\left(t\right),\psi\left(a,b\right)\left(t\right)\right)=\int_{-\infty}^{\infty}f\left(t\right)\frac{1}{a}\psi^{*}\left(a,b\right)\left(t\right)dt \tag{2.2}$$

Here, $C$ represents the wavelet coefficient at a single scale $a>0$ and position (in time) $b$. $\psi$ is the analyzing function, and $^*$ indicates a complex conjugate. $t$ represents time, and $f$ represents the signal being transformed. An analagous calculation can be performed for the frequency-domain signal. This calculation is repeated across a range of scales and positions to attain values for the two-dimensional plot [60].

These calculations were performed using the cwt command in MATLAB and a Morlet mother wavelet. This analyzing function is represented by [61]:

$$\psi\left(a,b\right)\left(t\right)=\frac{1}{a}\exp\left[i\omega_{0}\left(\frac{t-b}{a}\right)\right]\exp\left[-\left(\frac{t-b}{a}\right)^{2}/2\sigma^{2}\right] \tag{2.3}$$

Here, $\omega_0$ is the frequency, and $\sigma$ is a measure of spread.

This calculation was performed on the linear-scaled, complex-valued S21 parameter data described in Section 2.2. Generally, the CWT plot, known as a scalogram, is displayed as the magnitude squared at each location. However, due to poor visualization using this method, a natural logarithm was performed on each component's magnitude square. This increased the amount of visible information for each plot. 12 plots were produced for each object-orientation combination, corresponding to all 12 trials. The 12 plot values for each object-orientation combination were then averaged at each location in the plot. These averages are shown in Fig. 2.6. For the empty chamber, 12 CWT plots from each of three days of data collection were produced by the same method as for the object-orientation plots. The 12 empty-chamber CWT plots were averaged for each day, and these are also shown in Fig. 2.6.

Calibration drift over the course of the data collection period is visible in the Empty Chamber calculations in Fig. 2.6. This was considered acceptable for the purposes of the experiment, as slight variation between measurements increased the difficulty of the machine learning task in Section 2.4. For the manual identification plots shown in Fig. 2.6, however, a best representative example for each object was chosen.

As in Section 2.3.1, objects with large RCSs were identifiable. For example, strong reflections are visible for the 0 and 90° orientations for Object #1 just after the 0.05 microsecond time point. Many object-orientation combinations yield identifying characteristics in the 0.05-0.075 microsecond, sub-1 GHz window of the plots. Most object-orientation combination's signals were overwhelmed by the background signal, though remained more readily identifiable than in Section 2.3.1. Despite this relative success, this classification task is excessively tedious and is not scalable to a large number of objects due to the small variation between plots at different radar targets.

Also note that this was performed on ideal data, as discussed in Section 2.3.1. Thus, in the real world, identifying characteristics are expected to be even more difficult to locate. This motivates the development of a more robust, machine learning-based approach to the radar classification task.

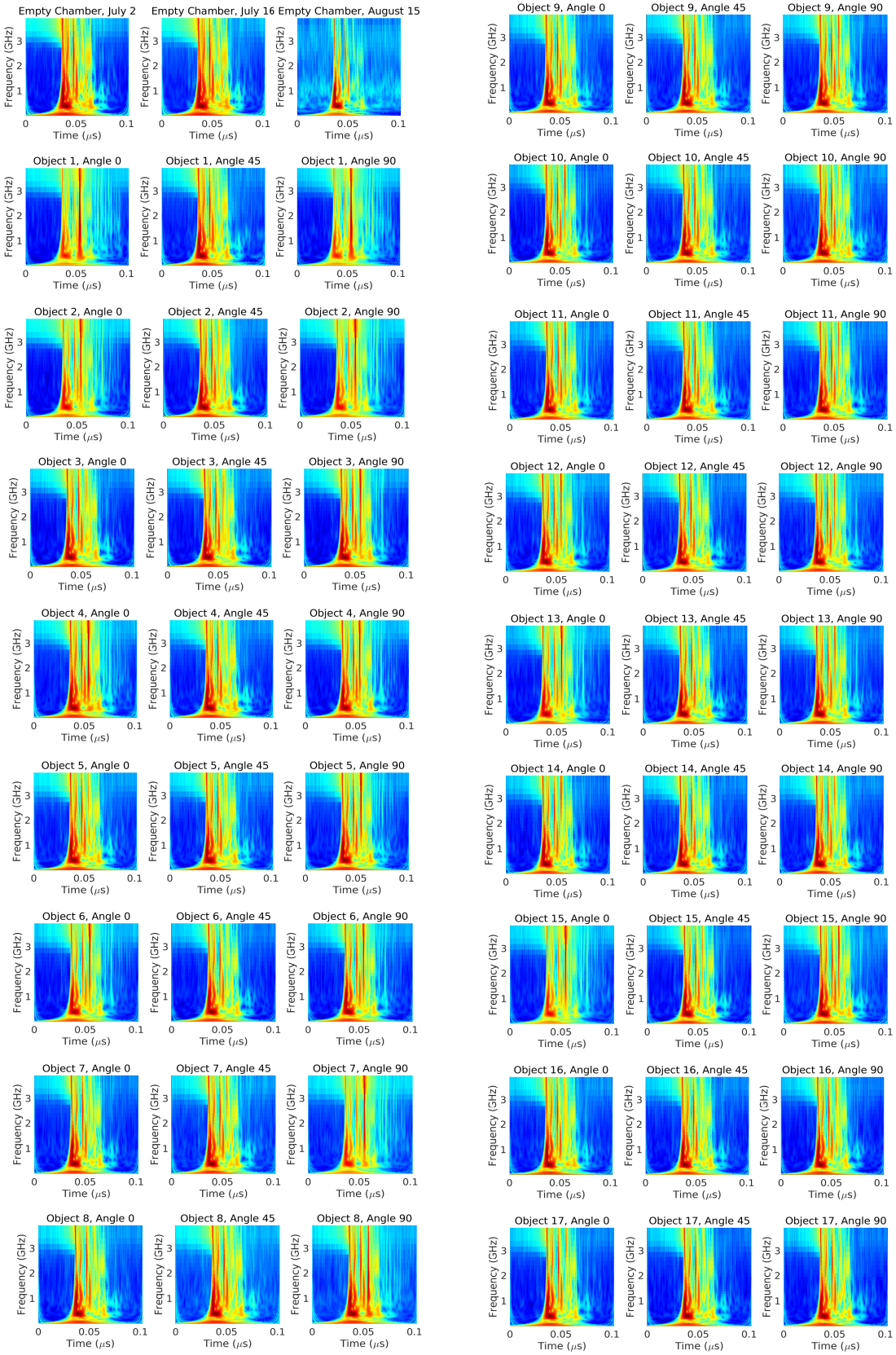


**Figure 2.6:** The CWT scalograms for each target-orientation combination are shown. This figure was produced by Greg Schuette.

## 2.4 Data Analysis: Machine Learning

Section 2.3 emphasizes the need for a robust, computer-based method of ATR. Several traditional network architectures were thus used to gauge their viability for the task. The success of these were then compared to the results of a CMSN. The work in this section was primarily performed by Khalid Youssef, and it is adapted from [46]. Note that only objects – not orientations – were identified using these machine learning approaches.

### 2.4.1 Data Preparation

Minimal data preprocessing was performed on the dataset [46]. This is to emphasize the ability of CMSNs to perform end-to-end classification, but a normalization process was still performed. First, the real and imaginary components of each trace were independently centered at zero by subtracting their mean. These components were then independently divided by their respective standard deviation. Note also that only the first 1600 of 1601 points in each trace were used.

Twelve data samples were produced for each object by combining the traces of each orientation into two-dimensional matrices. These were of length 1600 in one dimension – the length of each trace – and of length 6 in the other dimension – three orientations, with the real and imaginary portions placed in separate rows. This is shown at the top of Fig. 2.7.

### 2.4.2 Traditional Machine Learning Techniques

Four traditional machine learning techniques were used as a comparison to CMSN. These were: CNN, CNN Comittee, FCN, and FCN Comittee [46].

**Convolutional Neural Network** A CNN architecture containing four convolutional layers was used. Each convolutional layer is followed by batch normalization, a rectified linear unit (ReLU) activation, and a pooling layer, in that order. Following the final convolutional pooling layer is a hidden fully-connected layer. This is followed by an output layer with

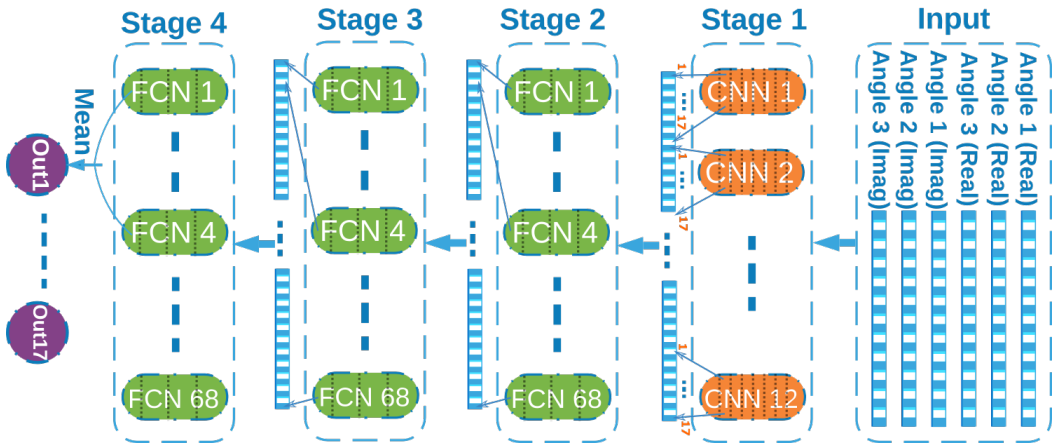


**Figure 2.7:** The CMSN structure used in this paper is shown. An example input data object is shown at the top. For any individual object, one trace from each of the 0, 45, and 90° orientations are grouped into one sample; these are named Angle 1, Angle 2, and Angle 3, respectively. The real components of each orientation are placed side-by-side, as are the imaginary components of each orientation. The block of real values is then placed side-by-side with the block of imaginary values to produce the two-dimensional input data of size 6x1600. This is then fed into twelve CNN nodes in the first stage. Each of these is trained to classify the radar target, resulting in 17 outputs each. The 17 outputs (from the softmax layer) from each of the 12 CNNs are concatenated to yield a stage output of size 204. This is inputted to the second stage, which is composed of 68 MLPs (labeled FCN in the figure). Each of these has an output of size 1, and these are concatenated to produce a stage output of size 68. This same process continues through the third and fourth network stages. The output at the fourth stage is grouped by the radar target each MLP attempts to identify. These values are averaged, yielding one value for each class. The value closest to 1 is chosen as the 'winner,' and the corresponding radar target is identified. This figure was produced by Khalid Youssef.

17 neurons – due to 17 radar target classes – and corresponding softmax and classification layers. The CNN was trained for a maximum of 50 epochs using the ADAM optimizer, with additional stopping criteria depending on validation error.

**CNN Committee** Twelve CNNs identical to those described above were trained on the dataset. Each CNN was initialized independently, so different local minima in the loss function should be located and different final network parameters located. When classifying data, each network was run independently, and a majority vote determined the committee's classification decision.

**Fully-Connected Network** The FCN contained two hidden layers, each with 200 neurons. The output layer consisted of 17 neurons followed by the softmax activation function and a classification layer. 100 epochs of ADAM were used to train the FCN. However, network parameters were saved after each epoch, and the network parameters yielding minimum validation loss were chosen for the network weights.

**FCN Committee** The FCN Committee consisted of twelve FCNs identical to the one described above. As with the CNN Committee, each network was independently initialized, and a majority vote determined the committee's classification decision.

### 2.4.3 CMSN Used for ATR

Finally, a CMSN was used to classify the radar targets. The first stage consisted of twelve CNNs identical to those described in Section 2.4.2. However, each CNN in the CMSN was only trained for three epochs via the ADAM optimizer. The output from the softmax layer – not the classification layer – for each CNN was concatenated; this retains information otherwise lost when the classification layer chooses a value of 1 for one output node and a value of 0 for the other 16 output nodes. The output size of this stage is 12 (number of CNN nodes) x 17 (number of outputs at each CNN), or 204. This serves as the input to the

second stage of the CMSN.

The remaining CMSN stages are similar to the MSN described in Section 1.4. There were three of these stages, consisting entirely of MLPs. Four MLPs targeting each class were used in each of these stages, for a total of 68 nodes per stage. Therefore, the output at each stage, after concatenation, is of size 68. Each MLP contained two hidden layers with 10 neurons per layer.

Classification decisions were based on the fourth stage's output. Recall that four MLPs target each class in this stage. The output of these four MLPs were averaged to provide a single output value for each class. The output value closest to 1 is chosen as the 'winner,' and the network predicts that the corresponding class is represented by the data inserted to the network.

### 2.4.4 Results

The most difficult classification task – and the only discussed here – attempted to classify all 17 objects. 11 samples were used for training and 1 for testing per class across all methods. This was repeated twelve times so that each trace was used for testing exactly one time. With each of these data splits, five trials were perfomed. This yields 12x5 or 60 total trials. The mean percent validation accuracy is shown below for each network structure.

| Method | CMSN | CNN | FCN | CNN Committee | FCN Committee |
|---|---|---|---|---|---|
| Accuracy (%) | 99.10 | 63.95 | 14.30 | 63.37 | 18.27 |

## 2.5 Conclusion and Future Work

This work began by using manual classification methods on the S21 parameter for 51 object-orientation combinations. This data was collected in ideal conditions, yet classification success was limited. If environmental noise, moving targets, multiple objects within range, etc., are included, specific identification of objects will likely be nearly impossible via these methods.

Machine learning methods proved more robust. Specifically, the CMSN consistently outperforms other methods by large margins in difficult conditions. The best competing method, CNN, yielded results of 63.95% accuracy. This compares to CMSN, which maintained validation accuracy of 99.10%. In other words, the CNN was incorrect $\frac{100-63.95}{100-99.10} = 40.06$ times as often as CMST.

Additionally, real-world applications will include many more potential classes of radar target. Larger networks will be necessary to classify all of these successfully. Fortunately, the natural scalability of CMSN enables its use with an increasing number of radar target classes. This scalability is, in large part, due to the distributed training of MST, which allows a very large network to be trained with limited computational resources regardless of full network size. The other methods, however, will require an ever-increasing amount of RAM and computational power to perform individual iterations, as each update tracks information about every network parameter simultaneously. Thus, CMSN is proposed as a potential solution to fully-automate ATR.

Further work should include larger datasets and a larger number of classes. Additionally, more difficult conditions should be tested to investigate ways to make CMSN more robust to environmental noise, moving radar targets, and other variables.

# CHAPTER 3

# Using Data Pre-Processing and CMSN to Improve Robustness to Time Shifts in RF Data Identification

In its primary introductory paper in 2017, multi-stage training (MST) was used to train a multi-stage network (MSN) which identified radio frequency (RF) transmitters [35]. This paper was motivated by a desire to increase the security of devices which communicate in the RF frequency range [35]. These devices are insecure for a variety of reasons, but a device's ability to accurately identify another device with which it is communicating can lessen the potential of transmitting sensitive information to a false impersonator of the intended receiver.

Electronic devices can be impersonated with the proper software. Therefore, hardware abnormalities must be used to distinguish good and bad actors on a secure network. This is possible due to inexact qualities within each component of a transmitter; manufacturing tolerances allow each component to be imperfect, and the culmination of specific imperfections in a device result in a unique *fingerprint* in the RF signal [62].

Youssef et al., 2017 showed that an MSN can effectively distinguish between RF transmitter/receiver pairs based on their transmitted Orthogonal Frequency Division Multiplexing (OFDM) packets [35, 63]. However, this analysis was performed on a dataset collected with stationary devices in good conditions. To deploy a neural network capable of identifying bad actors in the real world, messier data must be effectively analyzed. To explore this, the same dataset as used in [35, 63] is utilized. However, the data is artificially shifted in the time domain to mimic the effects of an inexact data collection method which could occur in day-to-day use cases. Two methods are utilized to increase robustness to these shifts in time:

1) Data pre-processing via use of different discrete wavelet transforms, some of which are quasi-time-invariant, and 2) the use of the convolutional multi-stage networks (CMSN) to take advantage the convolutional neural nework's (CNN) inherent robustness to data shifting discussed in Section 1.2.2.

## 3.1 Summary of Prior Results and Description of Data

Many techniques have historically been used to analyze RF data. For example, automatic target recognition (ATR) is a field with much interest, with identification accuracies of targets on the well-known Moving and Stationary Target Acquisition and Recognition (MSTAR) database approaching 100% with modern algorithms [49, 50, 51, 52, 53, 54, 55, 56]. However, this dataset concerns X-band synthetic-aperture radar images [48]. Thus, the developed methods are valuable for certain military applications, though they remain disconnected from the anti-spoofing problem identified in [35, 63].

To better represent this challenge, an OFDM RF packet dataset was collected [35, 63]. Data was collected using six radios and two transmitters for a total of 12 transmitter classes to be identified [35, 63]. There are 1000 OFDM packets per class, each a complex-valued OFDM packet in the time domain 10,000 time points in length [35, 63]. Further information regarding the dataset can be found at [35, 63]. This dataset is used throughout this chapter of the thesis.

This dataset is relevant, as OFDM packets are used for wideband digital communications including WLAN Wi-Fi, 4G mobile communications, television broadcasts, and many other direct communication techniques [64, 65]. In this field, there has been work to use AI to classify signal types and perform and interpret channel coding, modulation, and parametric estimation [66, 67]. However, research directly applying machine learning to identify specific RF emitter devices remains sparse [35, 63]. Thus, the application of deep neural networks to this problem was novel.

This problem fundamentally seeks to extract the features which are unique from one

transmitter to the next due to slight hardware differences [35]. Thus, the propensity for an MSN to extract multiple features, as discussed in Section 1.4, provided a natural motive to apply these networks to this problem. The results were very promising, with the MSN trained via Levenberg-Marquardt significantly outperforming standard fully-connected network (FCN), CNN, and support vector machine (SVM) architectures [35, 63]. Additionally, data preconditioning proved valuable.

As discussed in Section 2.3.2, CWTs provide a two-dimensional representation of signals. This alternative representation of the data is useful for analyzing data in many situations, including by a neural network. Indeed, the CWT was successfully used to improve the classification results using a CNN front-end on an MSN in [35]. Impressively, this yielded 99.95% accuracy on the validation set when only 10% of the total data was used to train the network, and with only 6 of 12 classes used to train the first of three MSN stages [35]. This compares to a maximum of 98.7% accuracy for a standard MSN that used all 12 classes to train every stage. The best results across other methods trained with 10% data were 87.3% (CNN), 84.8% (FCN), 87.6% (SVM Pearson VII Universal Kernel), and 67.6% (SVM PolyKernel) [35, 63]. The success when using the CWT likely stems at least in part from the superior representation of data to the network. However, a CWT does not add information despite its expansion of the data's size, and the result of this transform is, in fact, redundant [60]. Thus, the CNN front-end was necessary to recompress the data prior to insertion to the MSN.

To further explore and improve on this, other methods of manipulating data representation to the network were investigated. These data transformations are less redundant than the CWT, permitting smaller first-stage nodes to be used without needing to omit most of the transformed data. Additionally, the same calculations were performed using minimally-altered time-domain data. These calculations were performed using the network architectures and training parameters described in Section 3.3.

## 3.2 Preparation of Data

As stated at the beginning of this chapter, the dataset used in this chapter is the same as used in [35, 63]. Each OFDM packet contains 10,000 datapoints in the time domain, and most of this information is not necessary to achieve good classification results. Rather, subsets of the data samples were used, and the rising edges of the OFDM pulse were located and aligned [35, 63]. This was done via a thresholding process [35, 63].

### 3.2.1 Data Thresholding

To threshold the data, the real components of each packet were analyzed alone. Specifically, the first point for each packet whose real portion was greater than or equal to 0.05 became the first data point in the collected subset. 0.05 was chosen, as it is significantly larger than the noise while also being small enough to locate the beginning of the rising edge [35].

For example, assume that a thresholded vector of length 500 was desired. Also assume that the first point in the original OFDM packet whose real component is greater than or equal to 0.05 is the $501^{\text{st}}$ index. Then the subset collected in the new vector is the data between and including the $501^{\text{st}}$ and $1000^{\text{th}}$ index of the original OFDM packet.

The data preparation in all sections in this chapter except 3.2.5 utilize this same thresholding process. This serves to locate the signal's rising edge of each OFDM packet, where the individual hardware characteristics most affect the signal. Thus, this thresholding process locates a subset of the data with a relatively large amount of information to use for the transmitter classification task.

### 3.2.2 Time-Domain Data

To produce the time-domain data, the RF data was first thresholded via the method described in Section 3.2.1. Vectors of length 64 were collected.

For the MSN data (Method I), the absolute value of each vector was taken. After this,

each trace was translated in time a total of 18 times. These correspond to shifts of -9, -8, -7, ..., -1, 1, ..., 7, 8, and 9 units from the original signal. This was done by iterating the following in MATLAB for each trace and shifting amount:

```
% Initialize the vector which will hold the artificially-shifted data
% Note: original_vector is a column vector
new_vector = zeros( size(original_vector, 1), 1);

% Shift to the left
if n < 0
    % Place the preserved data into the new vector
    new_vector(1:end+n) = original_vector(1-n:end);

    % Find the slope at the corresponding end
    m = original_vector(end) - original_vector(end-1);

    % Use the slope to simulate the data at the end, with some noise
    for i = 1:abs(n)
        % Randomly select a slope within 10% of the original slope's value
        m1 = m * (rand*.2 + 0.9);
        % Use this slope to choose a value for missing section of data
        new_vector(end+n+i) = original_vector(end) + m1 * i;
    end

% Shift to the right
else
    new_vector(n+1:end) = original_vector(1:end-n);
    m = original_vector(2) - original_vector(1);
    for i = 1:n
        m1 = m * (rand*.2 + 0.9);
        new_vector(n+1-i) = original_vector(1) - m1 * i;
    end
end
```

Here, n represents the amount to shift by. Note that traces with no artificial shifting remain in the dataset.

From this full set of data, six subsets were produced. Each of the six subsets contains data with different amounts of overall shifting. These are labeled 0, 1, 3, 5, 7, and 9. The $n^{th}$ subset contains all of the data shifted to each of -n, -n+1, ..., 0, ... n-1, and n positions. Note that this implies 'subset 0 $\subset$ subset 1 $\subset$ subset 3 $\subset$ subset 5 $\subset$ subset 7 $\subset$ subset 9 = full dataset.'

Data preparation for the CMSN (Method II) was similar to Method I. Again, the OFDM packets were thresholded as outlined in 3.2.1, and a vector length of 64 was chosen. This time, however, the shifting code was performed on the complex-valued data. This resulted in six complex-valued subsets, again with maximum shifts of 0, 1, 3, 5, 7, and 9 units. Finally, real-valued network input data of size 2x64 was produced by placing the real and imaginary components of each vector side-by-side.

Sample plots of the time-domain data is shown in Fig. 3.1.

### 3.2.3 Discrete Wavelet Transforms

As stated in Section 3.1, the two-dimensional CWT yields a redundant representation of the RF signal. This is useful for visualization purposes, and it was also shown to improve the classification performance of MSNs analyzing OFDM RF packets [35]. However, it generally requires a larger neural network to analyze this input due to its expansion of the data's size. In an attempt to improve the data's representation to the network while also keeping the necessary size of input nodes relatively small, the single-level, one-dimensional discrete wavelet transform (DWT) was investigated. The total output size of this transform is the size of the original signal plus 2, yet a signal can be completely reconstructed; the signal's information content is maintained in a non-redundant manner. The DWT's ability to improve MSN robustness to shifting is also investigated.

The DWT is similar to the Discrete Fourier Transform (DFT) [68]. The DFT decomposes

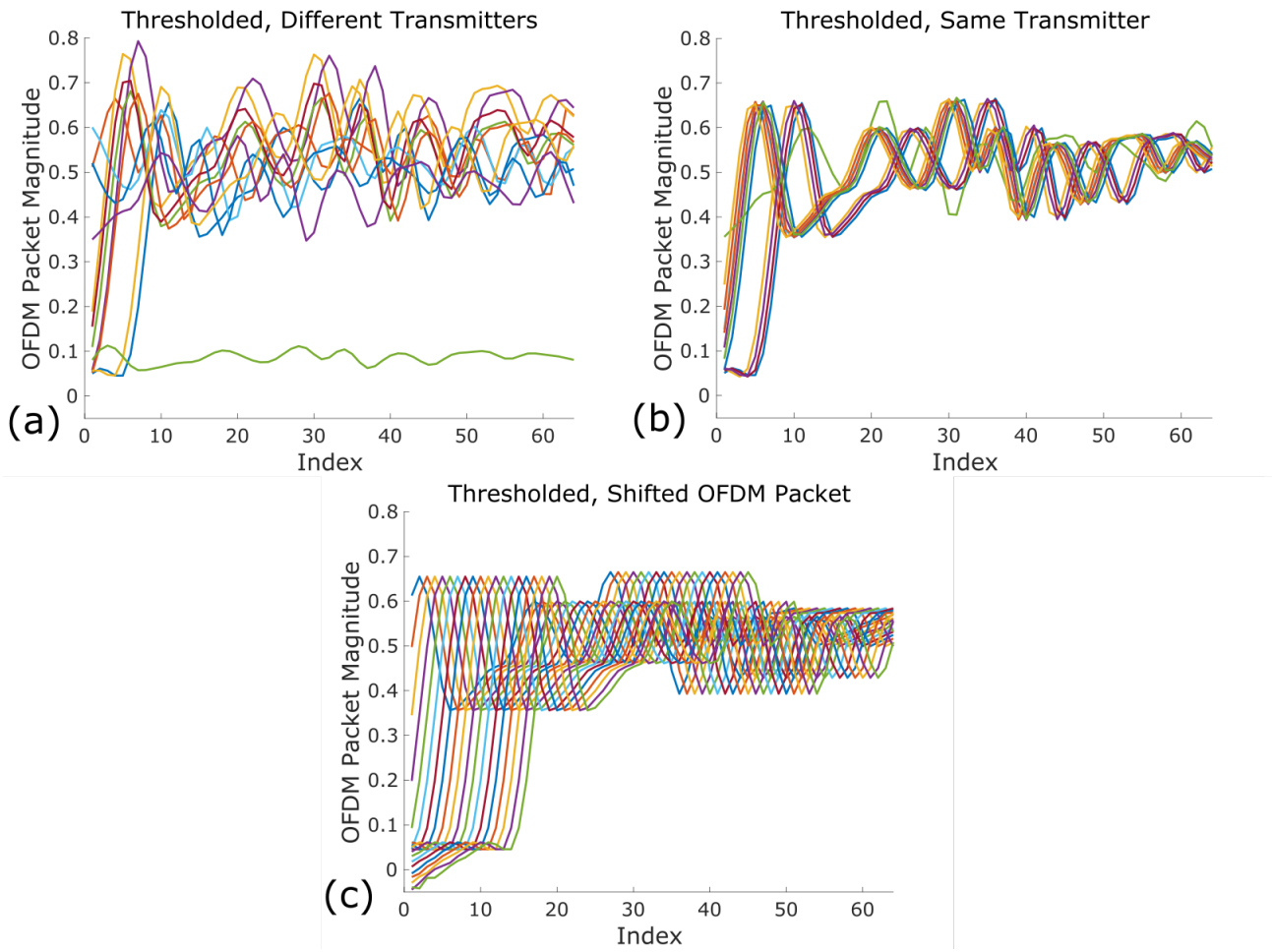


**Figure 3.1:** Three groupings of time-domain OFDM packets are shown. In each case, the signal's absolute value is shown. (a) One thresholded sample from each of the 12 transmitters is shown. Significant variance is visible. (b) 12 thresholded samples from a single transmitter are shown. Moderate variance between trials is visible. (c) One thresholded OFDM packet is shown with 19 different shifts in the time domain. These shifts are -9, -8, ..., 8, and 9. The same signal is clearly seen, though the time translation makes classification by neural networks more difficult.

a signal into sinusoidal basis functions, each dependent on a unique frequency [68]. The resulting transform can be perfectly transformed back into the original signal, meaning that no information is lost [68].

The DWT, however, decomposes a signal into a set of orthogonal wavelet basis functions [68]. Unlike the sinusoidal functions in a DFT, the wavelet functions are nonzero over only a subset of the signal's length [68]. These functions are derived from a common mother wavelet $\psi$ which is dilated, translated, and scaled [68]. In the DWT, each mother wavelet is paired with a *scaling* function $\phi$ to allow complete reconstruction, analagous to the Continuout Wavelet Tranformation (CWT) reconstruction shown in Eq. 3.1 [69].

$$x(t) = \sum_{n=-\infty}^{\infty} c(n)\,\phi(t-n) + \sum_{j=0}^{\infty}\sum_{n=-\infty}^{\infty} d(j,n)\,2^{j/2}\psi\left(2^j t - n\right) \tag{3.1}$$

Here, $x$ represents the original signal [69]. $j$ represents the *scale*, whose maximum value in the DWT is limited by the length of the input signal [69]. $c(n)$ represents the $n^{\text{th}}$ *scaling* (or *approximation*) coefficient, and $d(j,n)$ represents the $n^{\text{th}}$*wavelet* (or *detail*) coefficient at the $j^{\text{th}}$ scale [69]; the number of coefficients is also necessarily limited by signal length in the DWT. All data utilizing a standard DWT in this chapter utilizes the scaling coefficients and/or the first-scale wavelet coefficients, though higher scales are utilized in the MODWT and DTCWT, discussed in Section 3.2.4 and Section 3.2.5, respectively. These coefficients can be calculated by the discrete analogue of Eq. 3.2 and Eq. 3.3 [69].

$$c(n) = \int_{-\infty}^{\infty} x(t)\,\phi(t-n)\,dt \tag{3.2}$$

$$d(j,n) = 2^{j/2}\int_{-\infty}^{\infty} x(t)\,\psi\left(2^j t - n\right) \tag{3.3}$$

The mother wavelet used in this experiment was the Fejér-Korovkin filter. This was implemented using the `dwt` command in MATLAB, and the `'fk4'` command was used to specify a high-frequency Fejér-Korovkin filter for the single-level transformation.

The `dwt` command in MATLAB requires that the input be real-valued [70]. Additionally, the input to a DWT must be of length $2^N$ for some integer N [71]. Thus, to produce the DWT data, the RF data was first manipulated according to Method I in Section 3.2.2, except vector lengths of 512 ($2^9$) were chosen. This produces six subsets of the data, as described in Section 3.2.2.

Next, the `dwt` command in MATLAB was used to perform the DWT. The `dwt` command provides two outputs: Detail coefficients and approximation coefficients for the first-level decomposition [70]. Each of these is a vector of length $\frac{N}{2} + 1$, where N is the length of the signal placed into the function [70]. Because the data inputted to the function is 512 points long, each of the returned vectors is length 257, for a total of 514 datapoints after the wavelet decomposition.

64 datapoints from the DWT were chosen to create vectors of length 64 used as input to the neural networks. Specifically, the first 32 points from each of the detail and approximation coefficient vectors were concatenated.

The wavelet and approximation coefficients for several groupings of data are shown in Fig. 3.2.

### 3.2.4 Maximal Overlap Discrete Wavelet Transforms

The DWT was investigated for its ability to increase robustness to shifting in the time domain. However, the DWT is *not* generally invariant to shifts in time [71]. As such, the maximal overlap discrete wavelet transform (MODWT) was also investigated for its ability to improve robustness against shifting.

The mathematics of the MODWT is similar to that of the DWT. However, there are several important qualitative differences [71]. For the purposes of this study, the most important differences are (1) the MODWT is redundant, and (2) the MODWT is invariant to circular shifts of data [71]. The latter provides the potential to minimize alignment artifacts of the starting data – i.e. artifacts due to shifting in time of the RF data [71]. The

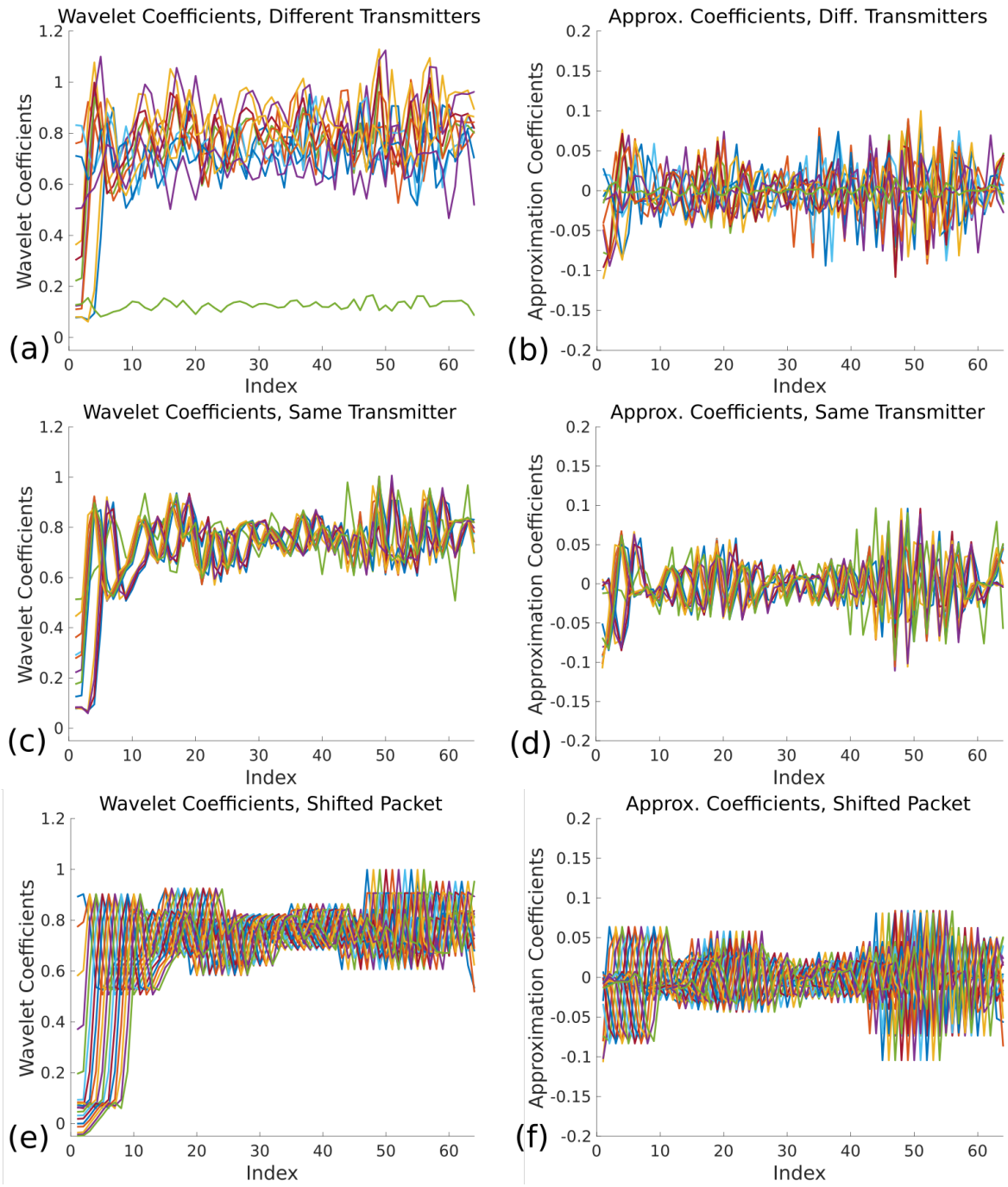


**Figure 3.2:** The wavelet and approximation coefficients for several groups of OFDM packets using the `dwt` command and `` `fk4' `` mother wavelet in MATLAB are shown. Though the total length of each vector is 257, only the first 64 values are shown. This is for two reason: The first 64 values are fed into the neural networks, and it also improves data visualization in this figure. (a), (c), and (e) show wavelet coefficients. (b), (d), and (f) show approximation coefficients. (a) and (b) show the results of one sample from each of 12 transmitter classes. Large variation is seen between samples. (c) and (d) show the results of 12 samples from a single transmitter class. A moderate amount of variation between samples is seen. (e) and (f) show the results of one sample shifted 19 times. This yields data for 19 DWTs, one shift for each of -9, -8, ..., 8, and 9 prior to transformation. Significant variance is seen. This is expected due to the lack of time invariance for the DWT.

former, however, decreases information density. This decreases the amount of information inputted to the MSN when equal vector lengths are compared.

Prior to performing the MODWT, the data was prepared by utilizing Method I described in Section 3.2.2. However, vectors of length 512 were utilized. This is, in part, because the MODWT requires real-valued input. Unlike the DWT, however, the MODWT does *not* require vectors of length $2^N$. Rather, the length 512 was chosen to maintain consistency with Section 3.2.3.

The MODWT was implemented using the `modwt` command in MATLAB. The `'fk4'` mother wavelet was chosen, as in Section 3.2.3. With an input of size 512 ($2^9$), the MODWT can perform a nine-level transformation. Thus, the `modwt` function returns a 10x512 matrix. The first nine rows provide the wavelet coefficients for the nine $2^k$ scales [72]. The tenth row provides the scaling coefficients for the $2^9$ scale [72].

Plots of these components are shown in Fig. 3.3. As shown, the ninth scale varies little depending on the data's shifting. Additionally, the scaling coefficients vary greatly depending on transmitter, yet are robust to shifting and are consistent within a transmitter class. Thus, these two components were chosen to construct network input data. To do this, vectors of length 64 were produced by concatenating the first 32 components of the scaling coefficient vector with the ninth-scale wavelet coefficients' $160^{\text{th}}$ to $181^{\text{st}}$ components, inclusive.

### 3.2.5 Dual Tree Wavelet Transforms

In another attempt to minimize the effects of shifting of the RF data in the time domain, the Dual Tree Complex Wavelet Transform (DTCWT) was utilized. This method benefits from improved (though imperfect) shift invariance. It is also less redundant than the undecimated DWT [73]; the undecimated DWT is very similar to the MODWT discussed in Section 3.2.4, so information is expected to be more compact with the DTCWT than the MODWT. This increased shift invariance and decreased redundancy provide rationale for potentially better results than in the other methods. These qualities are achieved by performing two independent multi-scale DWTs with specially-designed wavelet and scaling

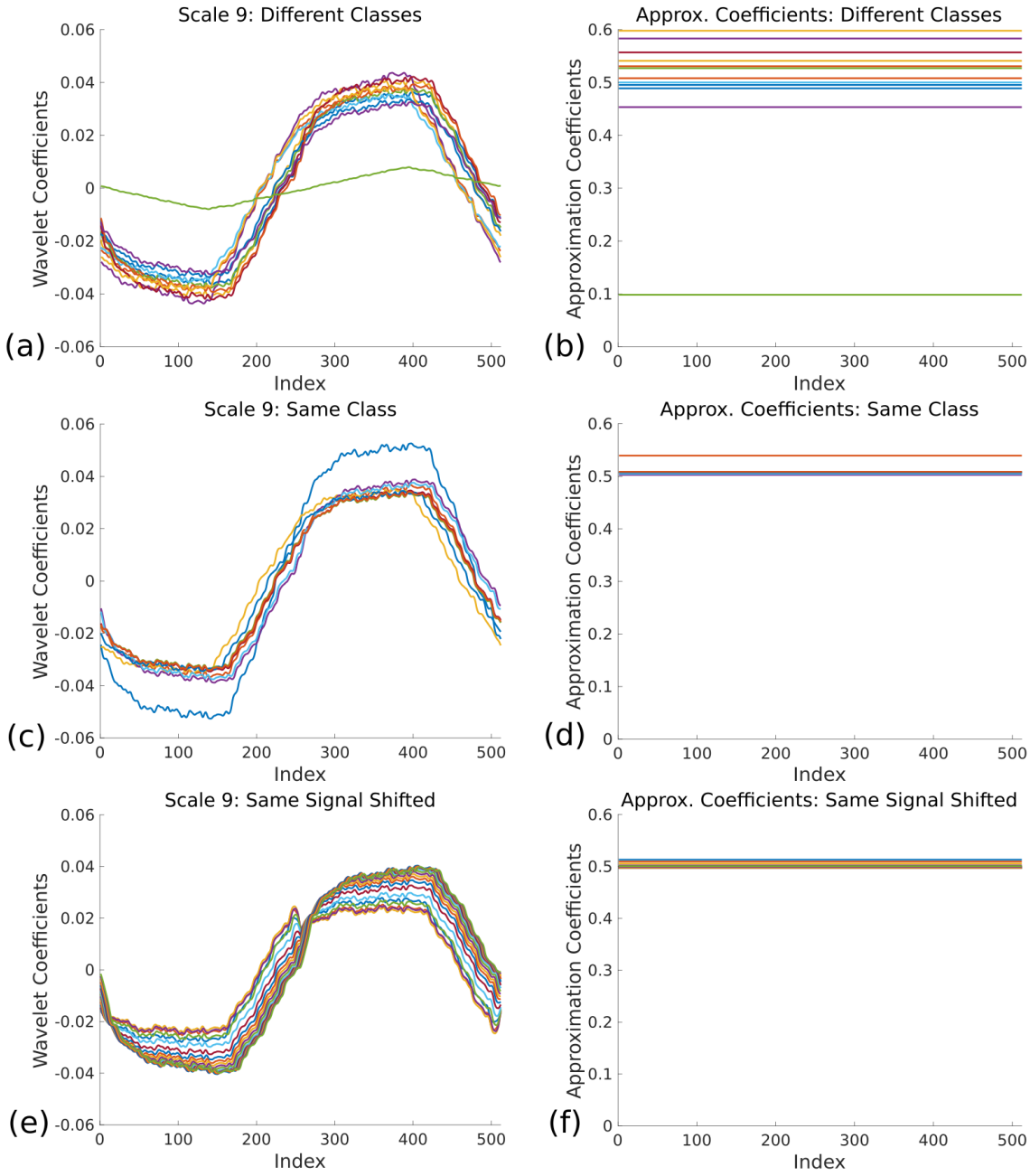


**Figure 3.3:** Plots of the ninth-scale wavelet coefficients and ninth-scale approximation coefficients for the MODWT using the `fk4' mother wavelet in MATLAB are shown. (a) The ninth-scale wavelet coefficients for one sample from each of 12 RF transmitter classes are shown. Signals appear similar between transmitters, though related features are shifted slightly with respect to the index. (b) The ninth-scale approximation coefficients for the same traces as in (a) are shown. Significant variation between transmitters is seen. (c) The ninth-scale wavelet coefficients for nine separate samples taken with the same transmitter are shown. Notable variation is seen, though there is less shifting along the index than in (a). (d) The ninth-scale approximation coefficients for the same traces as in (c) are shown. Little variation is seen with one exception. (e) The ninth-scale wavelet coefficients for a single sample translated in time is shown. A total of 19 curves are shown, corresponding to pre-MODWT time-translation shifts of -9, -8, ..., 8, and 9. Variation in magnitude is seen, though equivalent features appear at the similar indices. (f) The ninth-scale approximation coefficients for the same traces as described in (e) are shown. Little variation is seen with time translation.

filters.

The two multi-scale DWTs are known as *trees*, hence the name "Dual Tree" Complex Wavelet Transform [73]. This requires two distinct two-channel filter banks [73]. The wavelet and scaling functions of one tree must be the Hilbert transforms of the wavelet and scaling functions from the other tree [73]. These are known as *Hilbert pairs*, and they have the property that each frequency component in one can be generated by performing a quarter-shift on the frequency component of the other [69]. For example, Eq. 3.4 shows that cosine and sine functions form a Hilbert pair [69].

$$\cos\left(n\omega - \frac{\pi}{2}\right) = \sin\left(n\omega\right) \tag{3.4}$$

Additionally, the filters *within* a tree must obey the perfect reconstruction (PR) condition, whose sufficient conditions are shown in Eq. 3.5 and Eq. 3.6 [74].

$$\tilde{L}\left(e^{i\omega}\right) L\left(e^{i\omega}\right) + \tilde{H}\left(e^{i\omega}\right) H\left(e^{i\omega}\right) = 2e^{-iK\omega} \tag{3.5}$$

$$\tilde{L}\left(e^{i\omega}\right) L\left(e^{i(\omega+\pi)}\right) + \tilde{H}\left(e^{i\omega}\right) H\left(e^{i(\omega+\pi)}\right) = 0 \tag{3.6}$$

Here, $H$ and $L$ correspond to the high- and low-pass filters, respectively [74]. High- and low-pass filters correspond to the wavelet and scaling functions, respectively. Therefore, filters cannot be arbitrarily chosen [73].

The final output of the DTCWT is complex [73]. This results by combining the wavelet coefficients from each tree, where one tree provide the real components and the other tree provides the imaginary components of the final output [74, 73].

Unlike with the other methods, the OFDM packets were not thresholded prior to performing the DTCWT. However, the signal being transformed must still be real-valued [75]. As such, the absolute value of the entire OFDM packet of 10,000 is taken; like the MODWT, the DTCWT does not require lengths of $2^{\mathrm{N}}$. This vector of length 10,000 is then shifted in

time in the same manner as described in Section 3.2.2. These modified traces are then used to produce equivalent data subsets as described in Section 3.2.2.

The DTCWT was implemented using the `dualtree` command in MATLAB. This specifically corresponds to the Kingsburgy Q-Shift 1-Dimensional Dual-Tree Complex Wavelet Transform [75]. This name comes from the use of the orthogonal Q-shift Hilbert wavelet in use; this filter is of length 10 and satisfies the required Hilbert transform condition [75].

As implemented for this thesis, the `dualtree` command returns three objects. The first is a list of the final-level approximation coefficients, which are ignored. The other two are cells of size 13x1. These correspond to the 13-level transform. ($2^{13} = 8192$ is the largest power of 2 which remains less than 10,000, or the length of the input signal.) The first of these cells contains the wavelet coefficients at each level. These values are complex, with the real and imaginary components coming from the different trees [75]. The second cell contains the approximation coefficients at each level. These are real-valued. Sample plots for the DTCWT are shown in Fig. 3.4.

After transforming each signal, the seventh level was chosen to produce the MSNs' inputs. For the MSNs, the absolute value of the first 32 wavelet coefficients and the unmodified values of the $11^{\text{th}}$ to $42^{\text{nd}}$ approximation coefficients were concatenated. These indices were chosen for their consistency within transmitter classes and with shifting, but relatively large variance between transmitter classes. This can be seen in Fig. 3.4. For comparison, equivalent plots for the second scale are shown in Fig. 3.5. Similar results are noted for both scales.

For the CMSN data, the absolute value of the first 64 wavelet coefficients were placed side-by-side with the unmodified $11^{\text{th}}$ to $74^{\text{th}}$ approximation coefficients, producing a real-valued 2x64 neural network input.

## 3.3 Multi-Stage Training Applied to Time-Shifted RF Data

Two network architectures were used when investigating the effectiveness of the different data preparation methods. The first is a traditional MSN, similar in size to that used in [35,

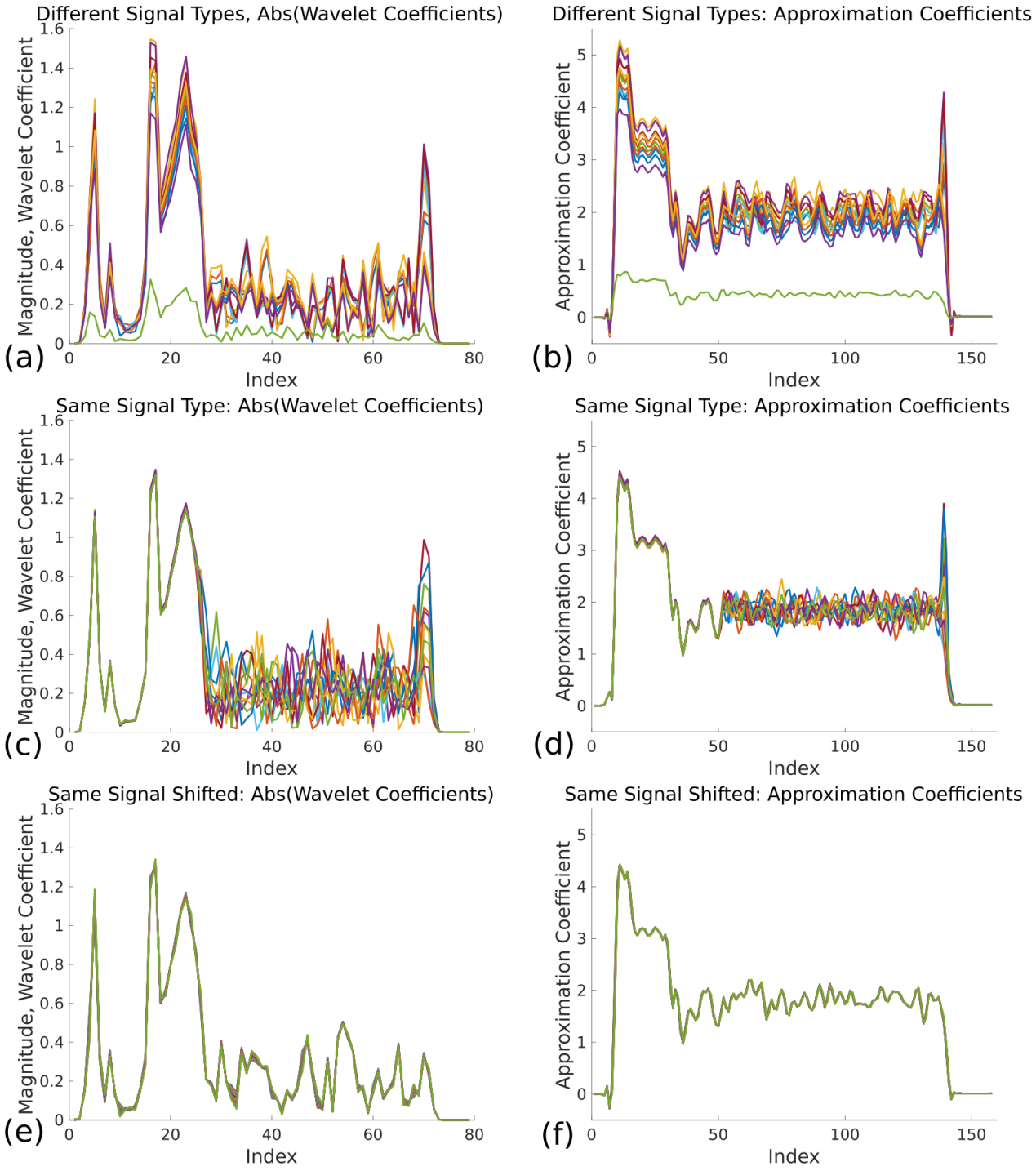


**Figure 3.4:** Data utilizing the DTCWT are shown. (a), (c), and (e) show the absolute value of the seventh-scale wavelet coefficients for the DTCWT for different groups of data. (b), (d), and (f) show the seventh-scale approximation coefficients for the DTCWT of different groups of data. (a) and (b) use the same group of data. Here, one OFDM packet from each of twelve transmitter classes is used to perform the DTCWT, and the results are shown. Significant variation by class is seen at low indices. (c) and (d) use 10 samples from the same transmitter class. Little variation between trials is seen at low indices. (e) and (f) use 19 examples versions of the same OFDM packet. Prior to the DTCWT, this packet is shifted 18 times, and one unshifted sample is maintained. This results in one sample shifted -9, -8, ..., 8, and 9 times. Significant invariance to these shifts is noted.

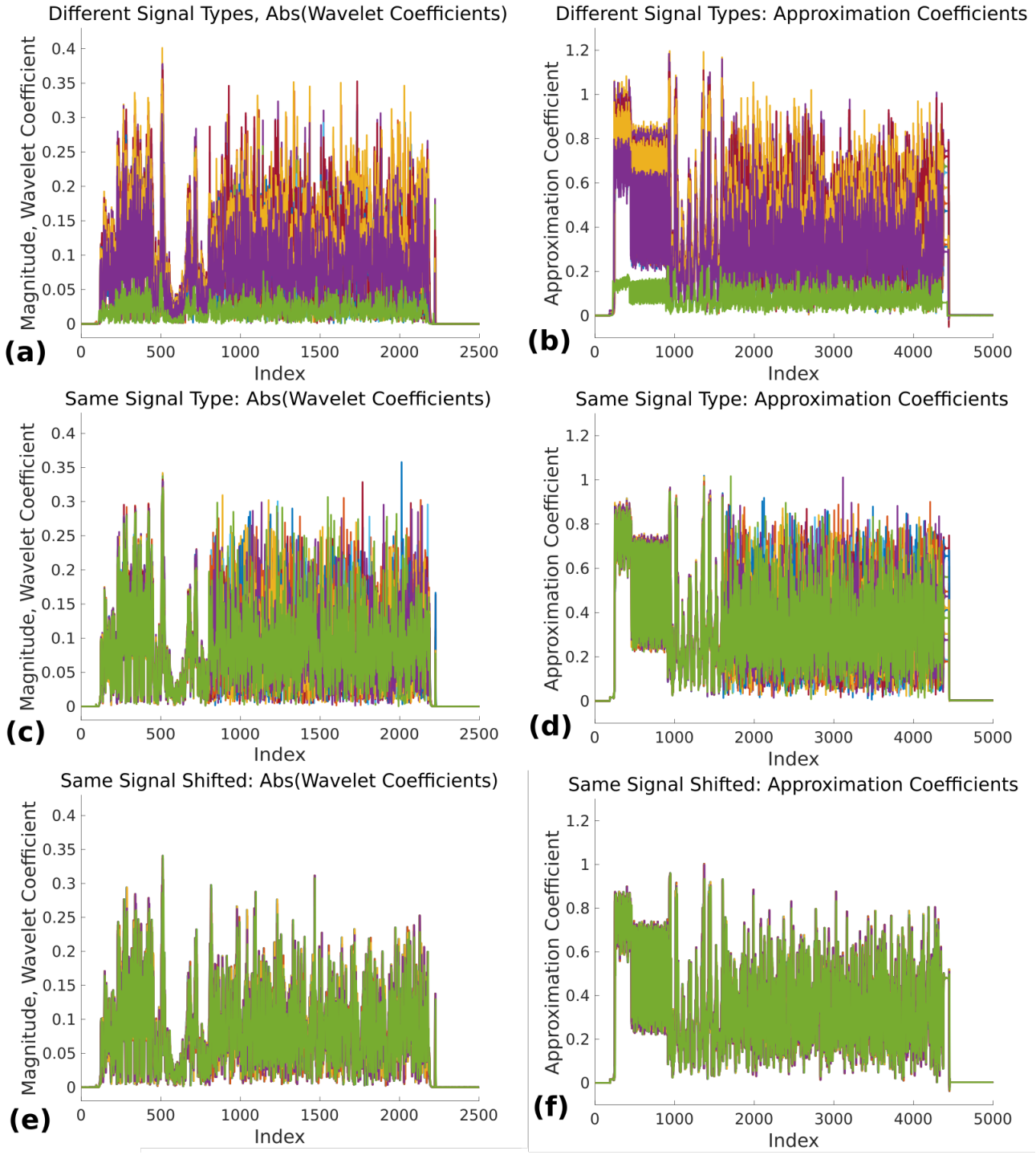


**Figure 3.5:** Data utilizing the DTCWT are shown. (a), (c), and (e) show the absolute value of the second-scale wavelet coefficients for the DTCWT for different groups of data. (b), (d), and (f) show the second-scale approximation coefficients for the DTCWT of different groups of data. (a) and (b) use the same group of data. Here, one OFDM packet from each of twelve transmitter classes is used to perform the DTCWT, and the results are shown. Significant variation by class is seen at low indices. (c) and (d) use 10 samples from the same transmitter class. Little variation between trials is seen at low indices. (e) and (f) use 19 examples versions of the same OFDM packet. Prior to the DTCWT, this packet is shifted 18 times, and one unshifted sample is maintained. This results in one sample shifted -9, -8, ..., 8, and 9 times. Significant invariance to these shifts is noted.

63]. This architecture was used on each of the datasets. The second network architecture was a CMSN. Calculations were performed on the time-domain data to both serve as a baseline when evaluating the effectiveness of the data preparation methods and to compare the effectiveness of the CMSN versus MSN with minimal data preparation.

### 3.3.1 MSN Structure

Each MSN used had identical architectures. The second and third stages of this MSN structure follow the description from Section 1.4, though the first stage was performed in a new way. The network contained three stages in total.

The first stage consisted of 60 FCNs, five per class. Initially, many nodes in this stage would learn to output 0.0833 (1/12, the number of classes) regardless of network input. Most likely, this indicates that the corresponding local minimum in the loss function was easy to find. This hindered network performance. To solve this problem, each node was permitted two outputs. These would target [1,0] or [0,1] if the target was the desired class or some other, respectively. In other words, network targets of 1 were replaced by [1,0], and network targets of 0 were replaced by [0,1]. Each node had two hidden layers, each with 15 neurons. The output size required an two output neurons. Also, the output neurons were followed by the cross-entropy loss function. MATLAB built-in functions do not permit jacobian training with the cross-entropy loss function, as the assumption made in Eq. 1.40 is not valid in that case. As such, these nodes were trained with the scaled conjugate gradient training algorithm. Each FCN was trained with 100 iterations, though training was stopped prematurely if loss decreased below 0.0001.

FCN outputs in the first stage always add to 1 due to the use of the softmax function on the network outputs. As such, keeping both outputs for any sample is redundant – one value can always be calculated exactly from the other. Thus, the first output at each stage was taken, and the second output discarded. This yields one output per FCN. These outputs were concatenated for a stage output of size 60.

The second stage consisted of 48 MLPs, four per class. The output stage consisted of 36

MLPs, three per class. Every MLP in the MSN contained two hidden layers, each with 15 neurons. Each hidden layer used the hyperbolic tangent transfer function.

The Levenberg-Marquardt training algorithm was used to train the second and third stages. MLPs in the second stage were trained for 150 iterations. MLPs in the third stage were trained for 200 iterations. Training was prematurely stopped if the sum-square error decreased below 0.00001, and 0.000001 for MLPs in the second and third stages, respectively.

For final classification, the outputs from final-stage nodes corresponding to the same class were averaged. This yields one output per class. The output closest to 1 is chosen as the 'winner,' and the corresponding transmitter is chosen as the output class.

When training the MSNs, 90% of the total dataset was used for training and 10% for validation.

### 3.3.2 CMSN Structure

Each CMSN calculation in this chapter was performed with one of two CMSN architectures, a larger and smaller network. Each consisted of four stages, one with CNN nodes and three with MST nodes.

The first stage contained CNN nodes. The input to each is 2×64. In an attempt to analyze the first dimension thoroughly, it is padded on each side. This brings the total size to 4×64. This then passes through three convolutional layers, each followed by a ReLU activation.

The first convolutional layer consists of three filters, each of size 2×16×1. This yields an output of size 3×49×3. The next layer consists of six filters, each of size 2×16×3. This yields an output of size 2×34×6. The final convolutional layer consists of twelve filters, each of size 2×16×6. This yields an output of size 1×19×12.

The final convolutional layer is followed by the fully-connected output layer. This has 12 neurons, one for each of 12 transmitter classes. This layer utilizes the softmax activation function.

The first convolutional layer contains $3\times2\times16\times1 = 96$ weights. The second contains $6\times2\times16\times3 = 576$ weights. The third contains $12\times2\times16\times6 = 2304$ weights. Finally, the fully-connected output layer contains $(1\times19\times12)\times12 = 2736$ weights. Each filter and each fully-connected neuron has one bias, adding 33 parameters total. Thus, each node has 5745 parameters, making these small enough to be easily trained via second-order methods.

Each of the twelve transmitter classes is assigned a unique integer identifier between 1 and 12. During training, the target for class $n \in \{1, ..., 12\}$ is a vector containing a 1 in the $n^{\text{th}}$ position and 0's elsewhere. The Levenberg-Marquardt training algorithm was initially chosen for this stage. This was perfomed identically as described in Section 1.3.4, except the updates were performed based on whether the cross-entropy loss decreased. This loss function helps emphasizes error in the position whose output should be 1. This helps ensure that a local minimum with an informationless, uniform output equal to 0.0833 (1/12) is not approached, which frequently occurred when using mean-square error. However, testing revealed that training the CNN frontend with stochastic gradient descent and the cross-entropy loss function yielded better results, so this was used instead.

The smaller network utilized 5 nodes in the first stage, while the larger network utilized 10 nodes in the first stage.

Each of the remaining stages consisted of MLPs. The smaller network architecture utilized 24 MLPs in each of these stages. This corresponds to 24 / 12 = 2 MLPs targeting each individual class at each stage. The larger network architecture utilized 48, 36, and 24 MLPs in the second, third, and fourth stage, respectively. This corresponds to 4, 3, and 2 MLPs per class per stage. For both network architectures and in each MLP stage, each MLP contained two hidden layers, each with 15 neurons. Training was performed with the Levenberg-Marquardt algorithm and mean-square loss function.

Following the final stage, the output for each MLP targeting the same class was averaged. This yields one network output for each class, 12 in total. The class whose output value is closest to 1 is chosen as the 'winner,' and the corresponding transmitter is decided to have produced the signal inputted to the network.

When training the CMSN, 10% of the total dataset was used to train the networks, and 90% was used for validation.

## 3.4 Results

The time-domain, DWT, MODWT, and DTCWT data preparation methods were all used to train MSNs of identical architecture. Increasing the amount of time-shifted data in the training set improved the validation accuracy outcome in every case. The DTCWT perfomed better than all other methods under most conditions. However, the MODWT performed best under the hardest condition – training with no shifting and validating with maximum shifting. The MODWT outperformed the DWT and time-domain data preparation methods more moderately under most conditions. The DWT outperformed the time-domain data modestly under most conditions. Under easier conditions – training with maximum shifting – classification accuracy was high for all data preparation methods and validation sets, though the DTCWT performed the best with 100% accuracy. Detailed results are shown in Tables 3.1, 3.2, 3.3, and 3.4.

The method generally yielding highest accuracy for time-shifted datasets was the DTCWT. As such, this method was chosen to train the CMSN as a comparison to the time-domain data. On time-domain data, the CMSN yielded better results than the MSN for time-shifted data. However, it underperformed compared to the MSN on unshifted data. When using the DTCWT data, the same trend is seen when comparing MSN to CMSN. These results are shown in Table 3.5.

## 3.5 Conclusion

The results in Tables 3.1, 3.2, 3.3, and 3.4 show that the DWT, MODWT, and DTCWT improve the MSN's robustness to time-shifting of OFDM packets during classification tasks. These results are confirmed when comparing Time-Domain 1 and DTCWT columns in Table 3.5, which correspond to identical networks and training schemes aside from the inputted

| | | Train | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 3 | 5 | 7 | 9 |
| Validation Set | 0 | 99.33(0) | 99.37(6) | 99.8325(0) | 99.5(2) | 99.46(6) | 99.8(2) |
| | 1 | 81(2) | 99.33(0) | 99.7487(0) | 99.5(1) | 99.5(1) | 99.7(1) |
| | 3 | 56.5(1) | 72(2) | 99.665(0) | 99.52(2) | 99.35(8) | 99.7(2) |
| | 5 | 45.5(7) | 56(3) | 85(1) | 99.40(5) | 99.36(9) | 99.6(2) |
| | 7 | 39(1) | 47(3) | 71.8(7) | 91.290(8) | 99.17(6) | 99.6(2) |
| | 9 | 35(2) | 41(2) | 61(9) | 79.1(9) | 89.37(2) | 99.4(2) |

**Table 3.1:** The results when using time-domain data with the MSN are shown. Two trials were performed for each value. The mean value is shown. The corresponding standard deviation is shown in parentheses, where the standard deviation's first significant figure appears in the last digit of the listed mean value. Different entries within a column correspond to using the same (trained) MSN to classify data in different validation subsets. The numbers at the top of a column and left of a row indicate which training and validation subset were used for the calculation, respectively. Section 3.2.2 discusses what is contained in each subset.

| | | Train | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 3 | 5 | 7 | 9 |
| Validation Set | 0 | 99.33(0) | 99.7(1) | 99.7(1) | 99.5(3) | 99.8325(0) | 99.539(6) |
| | 1 | 86(2) | 99.58(4) | 99.83(8) | 99.6(3) | 99.8046(0) | 99.6650(0) |
| | 3 | 67(2) | 82(1) | 99.719(9) | 99.5(2) | 99.78(6) | 99.69(2) |
| | 5 | 57(3) | 70(1) | 89.7(3) | 99.5(2) | 99.78(7) | 99.65(3) |
| | 7 | 50(3) | 62(2) | 79.54(9) | 93.2(2) | 99.63(6) | 99.60(2) |
| | 9 | 45(3) | 56(2) | 72.0(4) | 85.5(8) | 94.3(2) | 99.4799(0) |

**Table 3.2:** The results when using DWT data with the MSN are shown. Two trials were performed for each value. The mean value is shown. The corresponding standard deviation is shown in parentheses, where the standard deviation's first significant figure appears in the last digit of the listed mean value. Different entries within a column correspond to using the same (trained) MSN to classify data in different validation subsets. The numbers at the top of a column and left of a row indicate which training and validation subset were used for the calculation, respectively. Section 3.2.3 discusses what is contained in each subset.

| | | Train | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 3 | 5 | 7 | 9 |
| Validation Set | 0 | 99.3(3) | 99.7(1) | 99.4(3) | 99.46(6) | 99.3(3) | 99.29(6) |
| | 1 | 98.8(2) | 99.7(1) | 99.3(2) | 99.4137(0) | 99.4(2) | 99.30(8) |
| | 3 | 94.9(8) | 97.3(7) | 99.2(1) | 99.41(2) | 99.3(2) | 99.27(7) |
| | 5 | 88.5(1) | 90.3(9) | 94.9(7) | 99.26(1) | 99.3(2) | 99.26(5) |
| | 7 | 80.564(8) | 82(1) | 85(2) | 93.7(3) | 99.3(2) | 99.23(6) |
| | 9 | 73.2(3) | 75(2) | 76(4) | 85.87(7) | 96.6(4) | 99.19(5) |

**Table 3.3:** The results when using MODWT data with the MSN are shown. Two trials were performed for each value. The mean value is shown. The corresponding standard deviation is shown in parentheses, where the standard deviation's first significant figure appears in the last digit of the listed mean value. Different entries within a column correspond to using the same (trained) MSN to classify data in different validation subsets. The numbers at the top of a column and left of a row indicate which training and validation subset were used for the calculation, respectively. Section 3.2.4 discusses what is contained in each subset.

| | | Train | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 3 | 5 | 7 | 9 |
| Validation Set | 0 | 99.9162 | 99.9581 | 100 | 100 | 100 | 100 |
| | 1 | 99.9372 | 99.9581 | 100 | 100 | 100 | 100 |
| | 3 | 99.2462 | 99.9581 | 100 | 100 | 100 | 100 |
| | 5 | 91.6667 | 99.7069 | 100 | 100 | 100 | 100 |
| | 7 | 79.7948 | 95.8333 | 99.0787 | 100 | 100 | 100 |
| | 9 | 65.9966 | 77.3869 | 99.33 | 99.9868 | 100 | 100 |

**Table 3.4:** The results when using DTCWT data with the MSN are shown. One trial was performed for each value. Different entries within a column correspond to using the same (trained) MSN to classify data in different validation subsets. The numbers at the top of a column and left of a row indicate which training and validation subset were used for the calculation, respectively. Section 3.2.5 discusses what is contained in each subset.

| | | Trial | | | | |
|---|---|---|---|---|---|---|
| | | Time-Domain 1 | Time-Domain 2 | Time-Domain 3 | Time-Domain 4 | DTCWT |
| Validation Set | 0 | 91.26 | 93.58 | 96.35 | 95.18 | 98.68 |
| | 1 | 83.26 | 89.00 | 91.32 | 89.94 | 98.80 |
| | 3 | 67.19 | 74.35 | 75.06 | 74.40 | 98.56 |
| | 5 | 57.21 | 63.53 | 64.04 | 63.58 | 97.97 |
| | 7 | 50.83 | 56.25 | 56.91 | 56.32 | 96.64 |
| | 9 | 46.85 | 50.91 | 51.69 | 51.48 | 94.38 |

**Table 3.5:** Validation accuracies for three cases using the CMSN are shown. Each column corresponds to a single CMSN trained under different conditions. In each case, 10% of data was used to train the network, and no time-shifted data was included in the training set. Rows correspond to validation accuracies for the different validation subsets, described in Section 3.2.2 and 3.2.5. For columns Time-Domain 1, Time-Domain 3, and DTCWT, nodes were trained for 10, 15, 20, and 25 iterations each for the first, second, third, and fourth network stages, respectively. For Time-Domain 2, nodes were trained for 15, 100, 150, and 200 iterations each for the first, second, third, and fourth network stages, respectively. For Time-Domain 4, nodes were trained for 10, 50, 100, and 150 iterations each for the first, second, third, and fourth network stages, respectively. The CMSNs used in columns Time-Domain 1, Time-Domain 2, and DTCWT included 5, 24, 24, and 24 nodes for the first, second, third, and fourth nodes, respectively. The CMSNs used in columns Time-Domain 3 and Time-Domain 4 had 10, 48, 36, and 24 nodes in the first, second, third, and fourth stages, respectively. Each Time-Domain column represented networks trained with Time-Domain data prepared via Method II from Section 3.2.2. The DTCWT column corresponds to a network trained with the DTCWT data described in Section 3.2.5.

data. These improvements are minor with the DWT, moderate with the MODWT, and significant with the DTCWT. This order most likely stems from the near shift-invariance of the MODWT and DTCWT combined with the decreased redundancy of the DTCWT compared to the MODWT. However, it is unclear whether the additional benefit of DTCWT relative to the DWT, MODWT, and time-domain data is entirely a result of the DTCWT itself; not thresholding the data may be a significant factor. Therefore, future work should include performing the DTCWT calculations on data prepared with thresholding and shifting in an identical manner as the other methods.

The results also indicate that the CMSN provides an inherent improvement in robustness to time-shifting of the OFDM packets in classification tasks. This seems clear when comparing the classification results of the MSN architecture – the previous-best method on standard time-domain data – to the CMSN architecture. The CMSN yields noticeably better results on the shifted data despite significantly fewer training iterations and significantly less training data. This is clear when comparing the Time-Domain 1 and DTCWT columns in Table 3.5 to the 0 column in Tables 3.1 and 3.4, respectively. In the most difficult task for each of these – validation on a dataset including shifts up to magnitude 9 – the MSN yielded 35 and 66% validation accuracies for the time-domain and DTCWT data, while the CMSN yielded 46.85 and 94.38%.

However, the CMSN underperforms on unshifted data when compared to the MSN. The results in the Time-Domain 2 column of Table 3.5 reveal that simply increasing training iterations does not improve the unshifted results significantly. However, increasing CMSN size improved the unshifted results, as indicated in the Time-Domain 3 column of Table 3.5. Increasing training iterations did not improve results with the larger network, as indicated by comparing the Time-Domain 3 to the Time-Domain 4 columns in Table 3.5. The underperformance of CMSN in unshifted data can thus be attributed to one of three causes: 1) Limits the CMSN architecture, 2) poorly chosen CMSN hyperparameters, or 3) the use of a smaller amount of training data. Thus, to better understand how to improve CMSN results for effective RF identification, a CMSN should be trained with 90% training data

for a better comparison to the training method of the MSN, and more CMSN architectures should be tested.

# CHAPTER 4

# Selected Code

The preparation of this thesis required much coding. The most significant development, a generalized classification CMSN and MSN scheme, was programmed in Python using PyTorch. Several supporting functions were required for this, and they were included in the MSN class.

The generalized MSN class was developed to produce classification-oriented MSNs and convolutional CMSNs in PyTorch. This development is important, as these networks were previously tedious to implement, often requiring custom code when altering network architectures, training functions, etc. Furthermore, MSN and CMSN had not previously been implemented in PyTorch.

One of the supporting functions is the PyTorch implementation of the Levenberg-Marquardt training algorithm. Importantly, the implementation of Levenberg-Marquardt in PyTorch can be applied to CNNs. This is important, as MATLAB and PyTorch do not have this functionality built in.

Another key benefit to producing the general CMSN structure in PyTorch is that the code is written with a CUDA implementation selectable by the user. This allows the network to propagate and train on CUDA-enabled GPUs, significantly improving training times.

However, due to pending publications, this code is not included in the thesis. Rather, code for a generalized MSN scheme is shown written in MATLAB.

## 4.1 Code Description and Code

The following MSN code may be used to easily implement standard MST. Here, "standard" means that the description from Section 1.4 is followed.

The code in Section 4.1.1 has multiple restrictions. For example, the code only permits MLPs as nodes. Additionally, each MLP must have two hidden layers. The `train` function is MATLAB is used to train each node of the network. This limits the number of training functions available, though Levenberg-Marquardt may be used. The benefits of Levenberg-Marquardt discussed in Section 1.3.4 motivate the use of the `train` function over the alternative `trainNetwork` function. A list of valid training algorithms can be found at [76].

Despite these limitations, there are a number of generalized MSN characteristics for MSNs produced via the code in Section 4.1.1. Each generalized component is described in the comments of Listing 4.8 and the functions it calls.

Additional benefits in this code are the use of a GPU for training MLPs, when available. This can be seen in Listing 4.5. Additionally, multiple nodes within each stage are trained in parallel, further enhancing MSN training times. This can be seen in Listing 4.4, where the parfor-loop trains each node targeting an individual class in parallel. Finally, during forward computation of the MSN, nodes within a stage operate in parallel. In addition to this, each node performs its computation on a GPU, when available. These improvements enhance the rate at which the network operates, and the corresponding code is shown in Listing 4.2.

### 4.1.1 Code

First, the functions that perform forward computation of the MSN are shown. These functions are nested, with Lising 4.1 calling Listing 4.2

```
1 function [X] = MSN_forward(X,MSN)
2 % MSN_FORWARD This function operates an MSN in the forward direction.
```

```
% X represents input data, and it should be inserted to this function with
    % proper dimensions for the MSN frontend
% MSN Is a 1xN cell, where N is equal to the number of stages in the
    % MSN. Each element of the cell is another cell of size 1xM_i, ...
        where M is
    % the number of nodes in the i^th stage

for k = 1:length(MSN)
    X = MSN_stage_forward(X,MSN{1,k});
end

end
```

**Listing 4.1:** The MSN_forward function operates in a simple manner. An MSN is defined as a cell in MATLAB. This cell's length is equal to the number of stages. Each stage must be performed sequentially, and the output of one stage is the input to the following stage. As such, each stage is operated in the forward direction sequentially. This is generalized for number of stages.

```
function [out] = MSN_stage_forward(X,stage)
%MSN_STAGE_FORWARD This function performs the forward computation of an
%individual stage of an MSN, concatenating the output properly
% X is the input data to the network. It should already have proper
    % dimensions when inserted to this function
% stage is a 1xM cell, where M is the number of nodes in the stage. Each
    % element of stage is an individual MLP.

% Preallocate the output matrix.
out = zeros(length(stage),size(X,2));

try
    gpuArray(1) % Does not work if no GPU is available; sends to catch
    parfor k = 1:length(stage)
        out(k,:) = stage{1,k}(X,'useGPU','yes');
    end
```

```
17 catch
18     parfor k = 1:length(stage)
19         out(k,:) = stage{1,k}(X);
20     end
21 end
22
23 end
```

**Listing 4.2:** This code performs forward computation of a single stage of MSN. The output is concatenated to be directly prepared for insertion to the following MSN stage. The forward computation of a stage is parallelized, and computations are performed ona GPU, when available.

Next, code to initialize and train an MSN is shown. Again, these functions are nested, as Listing 4.3 calls Listing 4.4, and Listing 4.4 calls Listing 4.5.

```
1  function [MSN] = train_MSN(data,targets,MSN_Details)
2  %TRAIN_MSN This takes training data, training targets (as a vector of class
3  %labels as integers from 1:nClasses), and a cell containing the training
4  %data for stages/nodes
5  % The final output is an MSN, encoded as a 1xN cell, with N the number of
6  % network stages
7
8  % MSN_Details cell holds multiple components to train each stage
9      % MSN_Details{1,i} corresponds to the i^th stage of the network
10     % MSN_Details{1,i}{1,1}: nNodes     % int, for the i^th stage
11     % MSN_Details{1,i}{1,2}: train_frac % double, for i^th stage
12     % MSN_Details{1,i}{1,3}: train_fcn  % str, for each node at stage i
13     % MSN_Details{1,i}{1,4}: loss_fcn   % str, for each node at stage i
14     % MSN_Details{1,i}{1,5}: nNeurons   % int, for each node at stage i
15     % MSN_Details{1,i}{1,6}: nIt        % int, for each node at stage i
16     % MSN_Details{1,i}{1,7}: loss_goal  % double, for each node at ...
           stage i
17
18 % Initialize the MSN object
```

```
% length(MSN_Details) corresponds to the number of stages
MSN = cell(1,length(MSN_Details));

% Train the network/fill in all nodes, etc.
for k = 1:length(MSN)
    MSN{1,k} = train_stage(data,targets,MSN_Details{1,k}{1,1},...
        MSN_Details{1,k}{1,2},{MSN_Details{1,k}{1,3},...
        MSN_Details{1,k}{1,4},MSN_Details{1,k}{1,5},...
        MSN_Details{1,k}{1,6},MSN_Details{1,k}{1,7}});
    data = MSN_stage_forward(data,MSN{1,k});
end

end
```

**Listing 4.3:** This function can be directly called by a user to initialize and train an MSN, given the proper inputs. Descriptions for each input object are shown as comments at the top of the function. The function operates by training each stage sequentially via the train_stage function. The input data is then updated by performing a forward computation of the newly-trained stage. This iterates until the entire MSN is trained

```
function [stage] = train_stage(data,target,nNodes,train_frac,stageData)
%TRAIN_STAGE This function is used to train one stage of an MSN via MST
% data is the stage's input data
% train_frac is the fraction of training data that should be used to train
    % each node
% nNodes is a positive integer representing the number of nodes used in
    % this stage. Each node is assumed to be an MLP with two hidden layers,
    % and each hidden layer has nNeurons neurons per hidden layer. The
    % output layer should have one output
% target is a vector of target classes. Each class should be an integer
    % from 1 to nClasses, with every integer in between populated
% stageData is a cell with multiple components, corresponding to the
% train_node function:
    % stageData{1,1} = train_fcn % str
    % stageData{1,2} = loss_fcn  % str
```

```
16      % stageData{1,3} = nNeurons  % int
17      % stageData{1,4} = nIt       % int
18      % stageData{1,5} = loss_goal % double
19
20  % Calculate the number of classes that should be targeted
21  nClasses = length(unique(target));
22
23  % Check that a valid number of nodes were selected for this stage
24  if mod(nNodes,nClasses) ≠0
25      error(strcat('Improper number of nodes used in one or more stages.',...
26          '\nnNodes should be an integer multiple of nClasses'));
27  end
28
29  % Calculate the number of nodes targeting each class
30  nodes_per_target = nNodes / nClasses;
31
32  % Calculate the number of data samples which should be used for each node
33  % Default to train_frac of 1 if invalid value used
34  if train_frac ≤ 0 || train_frac > 1
35      nSamples = size(data,2);
36  else
37      nSamples = floor(size(data,2) * train_frac);
38  end
39
40  % Initialize the stage cell which will hold all nodes
41  stage = cell(1,nNodes);
42
43  %%%%%%%%%%%%%%%%%%%%%%%%
44  % Train each of the nodes
45
46  for k = 1:nClasses
47      % Create/update the tar object, which is a vector of 0's and 1's that
48      % serve as a target for an individual node attempting to target an
49      % individual class
```

```
50      clear tar
51      % Place 1's at target class, 0's elsewhere
52      tar = double(target == k);
53
54      n = (k-1)*nodes_per_target; % necessary to use parfor loop with the
55                                  % indexing of the cell
56      % Train all of the MLPs targeting this one class
57      parfor i = 1:nodes_per_target
58          % Shuffle the data by using an index
59          idx = 1:size(data,2);
60          idx = idx(randperm(length(idx)));
61          node = ...
                train_node(data(:,idx(1:nSamples)),tar(:,idx(1:nSamples)),...
62              stageData{1,1},stageData{1,2},stageData{1,3},stageData{1,4},...
63              stageData{1,5});
64          stage{1,n + i} = node;
65      end
66  end
67
68  %{
69  % m is a counter that will switch which class is targeted when appropriate
70  m = nodes_per_target;
71  target_class = 0;
72  for k = 1:nNodes
73      % Create/update the tar object, which is a vector of 0's and 1's that
74      % serve as a target for an individual node attempting to target an
75      % individual class
76      if m == nodes_per_target
77          clear tar
78          m = 0;
79          target_class = target_class + 1;
80          tar = double(target == target_class);
81      end
82
```

```
83      % Shuffle the data by using an index
84      idx = 1:size(data,2);
85      idx = idx(randperm(length(idx)));
86
87      % Initialize/Train the node and place in stage cell
88      stage{1,k} = train_node();
89
90      % Update m
91      m = m + 1;
92  end
93  %}
94  end
```

**Listing 4.4:** The train_stage function is shown. This function takes training data and targets for a given stage, as well as several stage architecture and training parameter commands. It returns a fully-trained stage. Note that this function shuffles data before training each node, and it allows subsets of the shuffled training data to be used at each node. Each node targeting the same class is trained in parallel. This can be easily modified to train nodes targeting any class in parallel. If, however, the copying of data to multiple workers is too memory-intensive, a sequential version of the code is placed as a comment at the bottom of the function. This may replace the prior for-loop.

```
1  function [MLP] = train_node(data,target,train_fcn,...
2      loss_fcn,nNeurons,nIt,loss_goal)
3  %TRAIN_NODE This function initializes and trains an MLP. It takes:
4  %training data; the targets for that training data; a training function
5  %declaration; a loss function declaration; a loss threshold to end training
6  %early; a maximum number of iterations to perform; a declaration of number
7  %of neurons per hidden layer.
8
9  %{
10 Assumptions:
11 Transfer function in hidden layers: tanh (implemented as tansig... default)
12 Transfer function in output layer: linear
13 Number hidden layers: 2
```

```
14 Number neurons output layer: 1
15
16 The first dimension of data should be the length of the input vectors
17 %}
18
19 % Prepare the MLP
20 MLP = feedforwardnet([nNeurons,nNeurons],train_fcn);
21 MLP.performFcn = loss_fcn;
22 MLP.trainParam.epochs = nIt;
23 MLP.trainParam.goal = loss_goal;
24 MLP.trainParam.showWindow = false;
25
26
27 % Initialize/Train the network
28 % Use GPU if possible
29 try
30     gpuArray(1); % Does not work if no GPU; sends to catch
31     MLP = train(MLP,data,target,'useGPU','yes');
32 catch
33     MLP = train(MLP,data,target);
34 end
35 end
```

**Listing 4.5:** The train_node function is shown here. This function takes training data and targets for an individual node and returns a trained MLP. Each MLP trained with this function has two hidden layers, each with a hyperbolic tangent transfer function. These are the default values for a feedforward network in MATLAB. An output layer with one neuron and a linear transfer function is also mandatory. Modifying this function can be done easily, however, if different MLP characteristics are desired.

Finally, a function used to find the single-value output of an MSN is shown. This is then used to produce a function which takes just data, a pre-trained MSN, and the number of classes to output a vector of identified classes.

```
1 function [votes] = vote(X,nClasses)
```

```
% VOTE This function takes data X (which should be the output of an MSN)
% and the number of classes nClasses. It uses this information to vote ...
    on a
% winning class (final network output).
% This function assumes that networks are trained so that outputs of the
% first (number of nodes in the final layer) / (nClasses) = N nodes target
% the first class, then nodes (N+1):2N target the second class, ...

% Preallocate a matrix to hold the AVERAGE outputs for each class for each
% data sample

avgs = zeros(nClasses,size(X,2));

% Find all the proper average values
nodes_per_class = size(X,1) / nClasses;
for k = 1:nClasses
    avgs(k,:) = mean(X(((k-1)*nodes_per_class+1):k*nodes_per_class,:),1);
end

% X is no longer needed. Delete to free RAM
clear X

% Preallocate the votes matrix
votes = zeros(1,size(avgs,2));

% Vote on the proper answers
% The value closest to 1 corresponds to the identified class in each case
% If two classes tie, the smaller class number is taken rather than
% outputting two values
for k = 1:size(avgs,2)
    temp = abs(avgs(:,k)-1);
    votes(k) = find(min(temp) == temp,1);
end

```

```
35 end
```

**Listing 4.6:** Here, the vote function is shown. This takes the output of an MSN and determines which class was identified by the network. It operates by calculate the average output at nodes targeting the same class. The average value closest to 1 is then selected as the identified class.

```
1 function [X] = MSN_operation(X,MSN,nClasses)
2 %MSN_OPERATION This function performs the forward compuation of the ...
      MSN. it
3 %then takes the output and uses the vote function to provide the identified
4 %classes. A row vector containing all identified classes is returned
5
6 X = MSN_forward(X,MSN);
7 X = vote(X,nClasses);
8
9 end
```

**Listing 4.7:** Here, the MSN_operation function is shown. This function uses an MSN, a data matrix, and an integer identifying the number of classes. It returns a row vector containing the class outputs for each data sample.

## 4.2 Using the Code

In this section, a toy problem is shown. This serves as an example so that the reader will understand how to call the `train_MSN` function to create an MSN with desired training parameters and MSN hyperparameters.

In this toy problem, 10 classes are produced, each representing the output values of 10 common functions over the domain $[0,\pi)$. Noise of up to magnitude 0.2 is randomly added to each point within each sample.

An MSN is then trained to identify a function based on the output. 10 nodes are used in each stage, with one node targeting each class. Each MLP contains five neurons in each of

two hidden layers. Each node may be trained for up to five epochs, though training is halted prematurely if mean-square error drops below 0.01, 0.001, or 0.0001 for the first, second, and third stage, respectively. The Levenberg-Marquardt training algorithm is used for each node within the network.

Nodes are trained with randomly-generated subsets of the training set. These subsets are 40%, 60%, and 80% of the total size of the training set for the first, second, and third stages, respectively.

Each criterion is inputted to the `MSN_Details` cell, which is then inserted to the `train_MSN` function. The `MSN_Details` cell has dimensions 1×N, where N is the number of stages desired in the network. The number of stages does not need to be specified anywhere else. Each element of the `MSN_Details` cell is another cell, this time of dimension 1×7. Each element of this cell is described as a comment in Listing 4.8.

The data object placed into the `train_MSN` function must be of dimension L×M. Here, L is the length of each data sample, and M is the number of data samples. Each column in this matrix is an independent data sample; data must be vectorized prior to being used in the `train_MSN` function. The targets object must be a 1×M vector of integers. Here, M is the same as before. Each element of the targets vector provides the class label of the corresponding column vector in the data matrix, e.g. `data(:,k)` has identity `targets(k)`.

In the following example, 100% accuracy on both training and validation sets is common. When calculating these accuracies, an example for using a pre-trained MSN is shown. For any data sample, the output of the MSN is an integer value associated with the class identified.

```
% Create 10 example classes; place in the data object

data = zeros(20,10000);
targets = zeros(1,10000);
% Sine
data(:,1:1000) = repmat(sin(0:.05*pi:pi*.99)',1,1000);
target(1:1000) = 1;
```

```
% Cosine
data(:,1001:2000) = repmat(cos(0:.05*pi:pi*.99)',1,1000);
target(1001:2000) = 2;
% Hyperbolic Tangent
data(:,2001:3000) = repmat(tanh(0:.05*pi:pi*.99)',1,1000);
target(2001:3000) = 3;
% Linear
data(:,3001:4000) = repmat((0:.05*pi:pi*.99)',1,1000);
target(3001:4000) = 4;
% -Linear
data(:,4001:5000) = repmat(-1*(0:.05*pi:pi*.99)',1,1000);
target(4001:5000) = 5;
% -Sine
data(:,5001:6000) = repmat(-sin(0:.05*pi:pi*.99)',1,1000);
target(5001:6000) = 6;
% -Cosine
data(:,6001:7000) = repmat(-cos(0:.05*pi:pi*.99)',1,1000);
target(6001:7000) = 7;
% -Hyperbolic Tangent
data(:,7001:8000) = repmat(-tanh(0:.05*pi:pi*.99)',1,1000);
target(7001:8000) = 8;
% ¬inverse
data(:,8001:9000) = repmat((1./(1 + (0:.05*pi:pi*.99)))',1,1000);
target(8001:9000) = 9;
% - ¬inverse
data(:,9001:10000) = repmat(-(1./(1+(0:.05*pi:pi*.99)))',1,1000);
target(9001:10000) = 10;

% Add noise to the data
data = data + rand(20,10000)*.4 - repmat(.2,20,10000);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Split into train/val sets (50/50) randomly
idx = 1:10000;
```

```
idx = idx(randperm(length(idx)));
trainData = data(:,idx(1:5000));
valData = data(:,idx(5001:end));
trainLabel = target(idx(1:5000));
valLabel = target(idx(5001:end));
clear data target

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Design the MSN

% Use 3 stages
MSN_Details = cell(1,3);

% MSN_Details cell holds multiple components to train each stage
    % MSN_Details{1,i} corresponds to the i^th stage of the network
    % MSN_Details{1,i}{1,1}: nNodes     % int, for the i^th stage
    % MSN_Details{1,i}{1,2}: train_frac % double, for i^th stage
    % MSN_Details{1,i}{1,3}: train_fcn  % str, for each node at stage i
    % MSN_Details{1,i}{1,4}: loss_fcn   % str, for each node at stage i
    % MSN_Details{1,i}{1,5}: nNeurons   % int, for each node at stage i
    % MSN_Details{1,i}{1,6}: nIt        % int, for each node at stage i
    % MSN_Details{1,i}{1,7}: loss_goal  % double, for each node at ...
        stage i

% Stage 1 Details
MSN_Details{1,1}{1,1} = 10;  % 3 Nodes per class -- 30 Nodes for Stage
MSN_Details{1,1}{1,2} = 0.4; % Use 40% of training data to train each node
MSN_Details{1,1}{1,3} = 'trainlm'; % Use Levenberg-Marquardt
MSN_Details{1,1}{1,4} = 'mse';     % Use mean-square error
MSN_Details{1,1}{1,5} = 5;         % 5 neurons/hidden layer at each node
MSN_Details{1,1}{1,6} = 5;         % up to 5 iterations at each node
MSN_Details{1,1}{1,7} = .01;       % Halt training if loss ≤ .01

% Stage 2 Details
```

```
75 MSN_Details{1,2}{1,1} = 10;  % 3 Nodes per class -- 30 Nodes for Stage
76 MSN_Details{1,2}{1,2} = 0.6; % Use 60% of training data to train each node
77 MSN_Details{1,2}{1,3} = 'trainlm'; % Use Levenberg-Marquardt
78 MSN_Details{1,2}{1,4} = 'mse';     % Use mean-square error
79 MSN_Details{1,2}{1,5} = 5;         % 5 neurons/hidden layer at each node
80 MSN_Details{1,2}{1,6} = 5;         % up to 5 iterations at each node
81 MSN_Details{1,2}{1,7} = .001;      % Halt training if loss ≤ .001
82
83 % Stage 3 Details
84 MSN_Details{1,3}{1,1} = 10;  % 3 Nodes per class -- 30 Nodes for Stage
85 MSN_Details{1,3}{1,2} = .8;  % Use 80% of training data to train each node
86 MSN_Details{1,3}{1,3} = 'trainlm'; % Use Levenberg-Marquardt
87 MSN_Details{1,3}{1,4} = 'mse';     % Use mean-square error
88 MSN_Details{1,3}{1,5} = 5;         % 5 neurons/hidden layer at each node
89 MSN_Details{1,3}{1,6} = 5;         % up to 5 iterations at each node
90 MSN_Details{1,3}{1,7} = .0001;     % Halt training if loss ≤ .0001
91
92 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
93 % Train the MSN
94 MSN = train_MSN(trainData,trainLabel,MSN_Details);
95
96 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
97 % Test the training and validation accuracies of the MSN
98
99 A = MSN_operation(trainData,MSN,length(unique(trainLabel)));
100 trainAccuracy = 100 * sum(A == trainLabel,'all') / length(trainLabel);
101 disp(strcat('Training Accuracy: ',string(trainAccuracy),'%'));
102
103 A = MSN_operation(valData,MSN,length(unique(trainLabel)));
104 valAccuracy = 100 * sum(A == valLabel,'all') / length(valLabel);
105 disp(strcat('Validation Accuracy: ',string(valAccuracy),'%'));
```

**Listing 4.8:** A script which solves the described toy probems is shown. Comments while building the MSN_Details cell describe what each element represents.

## 4.3 Future Work

The code in Section 4.1.1 is helpful for training standard MSNs in MATLAB. However, there are multiple functionalities not included that should be added.

It is currently easy to modify the architecture of MLPs by simply modifying the code in Listing 4.5. However, more freedom should be included so that MLP architecure – number of hidden layers, number of outputs per MLP, etc. – can be easily modified stage-by-stage. Additionally, the CMSN is not supported. This important development should be included to allow the benefits of a CNN stage to be realized in a user-friendly manner.

# Bibliography


[1] A. LeNail, "NN-SVG: Publication-ready NN-architecture schematics," *Journal of Open Source Software*, vol. 4, p. 747, January 2019. https://joss.theoj.org/papers/10.21105/joss.00747, accessed 2020-07-18.

[2] K. Youssef, N. N. Jarenwattananon, and L. Bouchard, "Feature-preserving noise removal," *IEEE Transactions on Medical Imaging*, vol. 34, no. 9, pp. 1822–1829, 2015.

[3] K. Youssef and L. Bouchard, "Feature-preserving noise removal," 2014. US Patent US9953246B2.

[4] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST database of handwritten digits." Yann LeCun, n.d. http://yann.lecun.com/exdb/mnist/, accessed 2020-07-03.

[5] Y. Shu, A. Hasenstaub, A. Duque, Y. Yu, and D. A. McCormick, "Modulation of intracortical synaptic potentials by presynaptic somatic membrane potential," *Nature*, vol. 441, pp. 761–765, April 2006. https://doi.org/10.1038/nature04720.

[6] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[7] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, Dec 1943.

[8] M. M. Khapra, "CS7015 (Deep Learning): Lecture 2." Indian Institute of Technology, Madras, January 2018.

[9] A. L. Chandra, "McCulloch-Pitts Neuron — mankind's first mathematical model of a biological neuron." Towards Data Science, July 2018. https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1, accessed 2020-07-03.

[10] B. Widrow and M. E. Hoff, "Adaptive switching circuits," in *1960 IRE WESCON Convention Record, Part 4*, (New York), pp. 96–104, IRE, 1960.

[11] M. Minsky and S. Papert, *Perceptrons*. MIT Press, 1969.

[12] A. Oppermann, "Activation functions in neural networks." Deep Learning Academy, n.d. https://www.deeplearning-academy.com/p/ai-wiki-activation-functions, accessed 2020-04-19.

[13] C. Dugas, Y. Bengio, F. Bélisle, C. Nadeau, and R. Garcia, "Incorporating second-order functional knowledge for better option pricing," in *Proceedings of the 13th International Conference on Neural Information Processing Systems*, NIPS'00, (Cambridge, MA, USA), p. 451–457, MIT Press, 2000.

[14] V. Sitzmann, J. N. P. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein, "Implicit neural representations with periodic activation functions," *arXiv*, 2020. https://arxiv.org/abs/2006.09661, eprint 2006.09661.

[15] E. Reingold and J. Nightingale, "History of neural networks." University of Toronto Course PSY371, 1999. http://www2.psych.utoronto.ca/users/reingold/courses/ai/cache/neural4.html, accessed 2020-07-08.

[16] Tutorials Point, "Supervised learning." Artificial Neural Network Tutorial, n.d. https://www.tutorialspoint.com/artificial_neural_network/artificial_neural_network_supervised_learning.htm, accessed 2020-04-19.

[17] A. Sakryukin, "Under the hood of neural networks. part 1: Fully connected." Towards Data Science, April 2018. https://towardsdatascience.com/under-the-hood-of-neural-networks-part-1-fully-connected-5223b7f78528, accessed 2020-07-03.

[18] M. Terry-Jack, "Deep learning: Feed forward neural networks (FFNNs)." Medium, April 2019. https://medium.com/@b.terryjack/introduction-to-deep-learning-feed-forward-neural-networks-ffnns-a-k-a-c688d83a309d, accessed 2020-07-07.

[19] B. Carremans, "Handling overfitting in deep learning models." Towards Data Science, August 2018. https://towardsdatascience.com/handling-overfitting-in-deep-learning-models-c760ee047c6e, accessed 2020-07-07.

[20] E. Culurciello, "The history of neural networks." Dataconomy, April 2017. https://dataconomy.com/2017/04/history-neural-networks/#:~:text=LeNet5,iterations%20since%20they%20year%201988!, accessed 2020-07-07.

[21] P. Ganesh, "Types of convolution kernels: Simplified." Towards Data Science, October 2019. https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37, accessed 2020-07-07.

[22] S. Saha, "A comprehensive guide to convolutional neural networks — the ELI5 way." Towards Data Science, December 2018. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53, accessed 2020-07-07.

[23] S. Dey, "CNN application on structured data – automated feature extraction." Towards Data Science, August 2018. https://towardsdatascience.com/cnn-application-on-structured-data-automated-feature-extraction-8f2cd28d9a7e, accessed 2020-07-07.

[24] V. Bushaev, "How do we 'train' neural networks?." Towards Data Science, November 2017. https://towardsdatascience.com/how-do-we-train-neural-networks-edd985562b73, accessed 2020-04-02.

[25] J. D. Olden and D. A. Jackson, "Illuminating the "black box": a randomization approach for understanding variable contributions in artificial neural networks," *Ecological Modelling*, vol. 154, no. 1, pp. 135 – 150, 2002.

[26] R. Parmar, "Common loss functions in machine learning." Towards Data Science, September 2018. https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23, accessed 2020-04-02.

[27] E. K. Chong and S. H. Żak, *An Introduction to Optimization*. John Wiley & Sons, Inc., 4 ed., 2015.

[28] H. Yu and B. Wilamowski, "Levenberg-Marquardt training," in *The Industrial Electronics Handbook*, vol. 5, ch. 12, pp. 12–1 – 12–16, CRC Press, 2 ed., January 2011.

[29] J. Dellinger, "Weight initialization in neural networks: A journey from the basics to kaiming." Towards Data Science, April 2019. https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79, accessed 2020-04-09.

[30] A. Beck, *First-Order Methods in Optimization*. Society for Industrial and Applied Mathematics and the Mathematical Optimization Society, 2017.

[31] J. Brownlee, "Understand the impact of learning rate on neural network performance." Machine Learning Mastery, January 2019. https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/#:~:text=The%20amount%20that%20the%20weights,range%20between%200.0%20and%201.0., accessed 2020-07-08.

[32] MathWorks$^{TM}$, "traingd," n.d. https://www.mathworks.com/help/deeplearning/ref/traingd.html, accessed 2020-07-28.

[33] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: Theory and applications," *Neurocomputing*, vol. 70, no. 1, pp. 489 – 501, 2006. Neural Networks.

[34] T. Terlaky, "Newton's method notes from continuous optimization algorithms." McMaster University, September 2001. http://www.cas.mcmaster.ca/~cs4te3/notes/newtons_method.pdf, accessed 2020-04-09.

[35] K. Youssef, L.-S. Bouchard, K. Z. Haigh, H. Krovi, J. Silovsky, and C. P. V. Valk, "Machine learning approach to RF transmitter identification." arXiv e-prints, 2017.

[36] M. T. Hagan and M. B. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, vol. 5, no. 6, pp. 989–993, 1994.

[37] R. Hauser, "Quasi-newton methods," in *Continuous Optimization*, vol. 4, Oxford University Computing Laboratory, 2006. http://www.numerical.rl.ac.uk/people/nimg/oupartc/lectures/raphael/lectures/lec4slides.pdf.

[38] J. Nachbar, "Definite matrices." Washington University in St. Louis, September 2014. https://pages.wustl.edu/files/pages/imce/nachbar/definitematrices.pdf, accessed 2020-04-09.

[39] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, pp. 431–441, June 1963.

[40] MathWorks$^{TM}$, "trainlm," n.d. https://www.mathworks.com/help/deeplearning/ref/trainlm.html, accessed 2020-07-07.

[41] A. Quesada, "5 algorithms to train a neural network." Neural Designer, n.d. https://www.neuraldesigner.com/blog/5_algorithms_to_train_a_neural_network#Levenberg-Marquardt, accessed 2020-04-19.

[42] K. Youssef and L. Bouchard, "Training artificial neural networks with reduced computational complexity," 2018. US Patent WO2019006088A1.

[43] P. Janpugdee, P. Pathak, P. Mahachoklertwattana, and R. Burkholder, "An accelerated DFT-MoM for the analysis of large finite periodic antenna arrays," *IEEE Transactions on Antennas and Propagation*, vol. 54, pp. 279 – 283, 02 2006.

[44] P. Janpugdee, P. H. Pathak, P. Nepa, and H. T. Chou, "A fast hybrid DFT-MoM for the analysis of large finite periodic antenna arrays on grounded substrates," *2002 USNC/URSI Radio Science Meeting*, June 2002.

[45] P. Janpugdee, P. H. Pathak, P. Mahachoklertwattana, and P. Nepa, "A fast hybrid dft-mom for the analysis of large finite periodic antenna arrays in grounded layered media," in *IEEE Antennas and Propagation Society International Symposium. Digest. Held in conjunction with: USNC/CNC/URSI North American Radio Sci. Meeting (Cat. No.03CH37450)*, vol. 4, pp. 7–10 vol.4, 2003.

[46] K. Youssef, Y. Cai, G. Schuette, D. Zhang, Y. Huang, Y. Rahmat-Samii, and L.-S. Bouchard, "Scalable end-to-end radar classification: A case study on undersized dataset regularization by convolutional-MST." unpublished.

[47] L. Doucet, "Iran plane crash: Ukrainian jet was 'unintentionally' shot down." BBC, January 2020. https://www.bbc.com/news/world-middle-east-51073621, accessed 2020-07-12.

[48] "MSTAR overview." The Sensor Data Management System, September 1995. https://www.sdms.afrl.af.mil/index.php?collection=mstar, accessed 2020-07-12, current as of 2019-02-26.

[49] X. Huang, Q. Yang, and H. Qiao, "Lightweight two-stream convolutional neural network for SAR target recognition," *IEEE Geoscience and Remote Sensing Letters*, pp. 1–5, 2020.

[50] H. Zhu, N. Lin, H. Leung, R. Leung, and S. Theodoidis, "Target classification from SAR imagery based on the pixel grayscale decline by graph convolutional neural network," *IEEE Sensors Letters*, vol. 4, no. 6, pp. 1–4, 2020.

[51] H. Zhu, W. Wang, and R. Leung, "SAR target classification based on radar image luminance analysis by deep learning," *IEEE Sensors Letters*, vol. 4, no. 3, pp. 1–4, 2020.

[52] Z. Sun, X. Xu, and Z. Pan, "SAR ATR using complex-valued CNN," in *2020 Asia-Pacific Conference on Image Processing, Electronics and Computers (IPEC)*, pp. 125–128, 2020.

[53] H. Dbouk, H. Geng, C. M. Vineyard, and N. R. Shanbhag, "Low-complexity fixed-point convolutional neural networks for automatic target recognition," in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1598–1602, 2020.

[54] S. A. Wagner, "SAR ATR by a combination of convolutional neural network and support vector machines," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 52, no. 6, pp. 2861–2872, 2016.

[55] F. Gao, T. Huang, J. Sun, J. Wang, A. Hussain, and E. Yang, "A new algorithm for SAR image target recognition based on an improved deep convolutional neural network," *Cognitive Computation*, vol. 11, no. 6, pp. 809–824, 2019.

[56] S. Chen, H. Wang, F. Xu, and Y. Jin, "Target classification using the deep convolutional networks for SAR images," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 54, no. 8, pp. 4806–4817, 2016.

[57] G. Quintero, J.-F. Zurcher, and A. K. Skrivervik, "System fidelity factor: A new method for comparing UWB antennas," *IEEE Transactions on Antennas and Propagation*, vol. 59, no. 7, pp. 2502–2512, 2011.

[58] D. Zhang and Y. Rahmat-Samii, "Top-cross-loop improving the performance of the UWB planar monopole antennas," *Microwave and Optical Technology Letters*, vol. 59, pp. 2432–2440, July 2007.

[59] H.-S. Lui and N. V. Shuley, "Joint time-frequency analysis of ultra wideband radar signals," *IEEE International Conf. on Signal Processing and Communication Systems*, pp. 1–8, 01 2007.

[60] MathWorks$^{TM}$, "Continuous wavelet transform and scale-based analysis," n.d. https://www.mathworks.com/help/wavelet/gs/continuous-wavelet-transform-and-scale-based-analysis.html, accessed 2020-07-12.

[61] A. Tangborn, "Wavelet transforms in time series analysis." University of Maryland, 2010. Course AOSC630, https://www2.atmos.umd.edu/~ekalnay/syllabi/AOSC630/Wavelets_2010.pdf, accessed 2020-07-12.

[62] S. U. Rehman, K. W. Sowerby, S. Alam, and I. Ardekani, "Radio frequency fingerprinting and its challenges," in *2014 IEEE Conference on Communications and Network Security*, pp. 496–497, 2014.

[63] K. Youssef, L. Bouchard, K. Haigh, J. Silovsky, B. Thapa, and C. V. Valk, "Machine learning approach to RF transmitter identification," *IEEE Journal of Radio Frequency Identification*, vol. 2, no. 4, pp. 197–205, 2018.

[64] Electronics Notes, "What is OFDM: Orthogonal frequency division multiplexing." Electronics Notes, n.d. https://www.electronics-notes.com/articles/radio/multicarrier-modulation/ofdm-orthogonal-frequency-division-multiplexing-what-is-tutorial-basics.php, accessed 2020-07-12.

[65] Keysight Technologies, Inc., "802.11 OFDM overview." Keysight Technologies, n.d. http://rfmw.em.keysight.com/wireless/helpfiles/89600B/WebHelp/Subsystems/wlan-ofdm/content/ofdm_80211-overview.htm, accessed 2020-07-12.

[66] Z. Zhao, M. C. Vuran, F. Guo, and S. Scott, "Deep-waveform: A learned OFDM receiver based on deep complex convolutional networks." arXiv e-prints, 2018.

[67] T. J. O'Shea, L. Pemula, D. Batra, and T. C. Clancy, "Radio transformer networks: Attention models for learning to synchronize in wireless systems." arXiv e-prints, 2016.

[68] M. C. Nechyba, "Introduction to the discrete wavelet transform (DWT)." University of Florida, February 2004. https://mil.ufl.edu/nechyba/www/eel6562/course_materials/t5.wavelets/intro_dwt.pdf, Course Material EEL6562.

[69] I. W. Selesnick, R. G. Baraniuk, and N. C. Kingsbury, "The dual-tree complex wavelet transform," *IEEE Signal Processing Magazine*, vol. 22, no. 6, pp. 123–151, 2005.

[70] MathWorks$^{TM}$, "dwt," n.d. https://www.mathworks.com/help/wavelet/ref/dwt.html, accessed 2020-07-14.

[71] D. B. Percival, "Maximal overlap discrete wavelet transform." University of Washington, 2018. https://faculty.washington.edu/dbp/s530/PDFs/05-MODWT-2018.pdf, Course Material for Stat/EE 530, accessed 2020-07-13.

[72] MathWorks$^{TM}$, "modwt," n.d. https://www.mathworks.com/help/wavelet/ref/modwt.html#buuu3sz-w, accessed 2020-07-14.

[73] MathWorks$^{TM}$, "Dual-tree complex wavelet transforms," n.d. https://www.mathworks.com/help/wavelet/ug/dual-tree-complex-wavelet-transforms.html, accessed 2020-07-13.

[74] D. Sundararajan, *Discrete Wavelet Transform*, ch. 14, pp. 247–264. John Wiley Sons, Ltd, 2015.

[75] MathWorks$^{TM}$, "dualtree," n.d. https://www.mathworks.com/help/wavelet/ref/dualtree.html, accessed 2020-07-15.

[76] MathWorks$^{TM}$, "fitnet," n.d. https://www.mathworks.com/help/deeplearning/ref/fitnet.html#bu2w2vc-1-trainFcn, accessed 2020-07-30.