AI Assisted Workflows for Computational Electromagnetics and Antenna Design

Oameed Noakoasteen

University of New Mexico - Main Campus

Follow this and additional works at: https://digitalrepository.unm.edu/ece_etds

Part of the Artificial Intelligence and Robotics Commons, Data Science Commons, and the Electrical and Computer Engineering Commons

Recommended Citation


This Dissertation is brought to you for free and open access by the Engineering ETDs at UNM Digital Repository. It has been accepted for inclusion in Electrical and Computer Engineering ETDs by an authorized administrator of UNM Digital Repository. For more information, please contact disc@unm.edu.
Oameed Noakoasteen

Candidate

Electrical and Computer Engineering

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Prof. Christos Christodoulou, Chair

Prof. Zhen Peng, Member

Prof. Edl Schamiloglu, Member

Prof. Jehanzeb H. Chaudhry, Member
AI Assisted Workflows for Computational Electromagnetics and Antenna Design

by

Oameed Noakoasteen

B.S., Zanjan University, 2009
M.S., K.N. Toosi University of Technology, 2013

DISSERTATION

Submitted in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy
Engineering

The University of New Mexico

Albuquerque, New Mexico

December, 2023
Dedication

To my parents, Behnaz and Hamid,
who steadfastly stood by me through every setback on my journey.
Acknowledgments

I would like to thank the following people:

My adviser Professor Christos Christodoulou for his guidance and unwavering support without which I would not have been able to complete this research.

Professor Zhen Peng for pointing me towards the field of AI/ML and for constructive discussions that shaped this research.

Fellow research group members, Shu Wang for introducing me to the Domain Decomposition Methods; Arjun Gupta and Jayakrishnan Vijayamohanan for the fruitful discussions, sharing of ideas and their datasets.
AI Assisted Workflows for Computational Electromagnetics and Antenna Design

by

Oameed Noakoasteen

B.S., Zanjan University, 2009
M.S., K.N. Toosi University of Technology, 2013
Ph.D. Engineering, University of New Mexico, 2023

Abstract

These days large volumes of data can be recorded and manipulated with relative ease. If valuable information can be extracted from them, these vast amounts of data can be a rich resource not just for the digital economy but also for scientific discovery and development of technology. When it comes to deriving valuable information from data, Machine Learning (ML) emerges as the key solution. To unlock the potential benefits of ML to science and technology, extensive research is needed to explore what algorithms are suitable and how they can be applied.

To shine light on various ways that ML can be impactful to engineering workflows, fields of computational EM and antenna design are chosen and efforts are focused on answering three fundamental questions: (1) Is it possible to obtain predictive models from the available simulation and measurement data? (2) What are the domain-specific machine learning algorithms required to convert various datasets to modeling knowledge? (3) Once the modeling knowledge has been learned, how to seamlessly incorporate it into a data-driven predictive environment?
Contents

List of Figures ix
List of Tables xi

1 Introduction 1

1.1 Artificial Intelligence . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Machine Learning . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.3 AI/ML Assisted Workflows . . . . . . . . . . . . . . . . . . . . . . . 6
1.4 Reproducibility . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2 Deep Learning 8

2.1 Feed-Forward Networks . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.2 Convolutional Layer . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.3 Recurrent Layer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.4 Deep FFNs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.5 Deep Learning Frameworks . . . . . . . . . . . . . . . . . . . . . . . 17
Contents

3 Emulation of the FDTD: RNNs 21

3.1 Finite Difference Time Domain .................................. 26
3.2 The Dataset ......................................................... 36
3.3 The Architecture .................................................. 37
3.4 Numerical Experiments ............................................. 39
3.5 ERD Model and DDM Parallelization ............................... 42

4 Emulation of the FDTD: Transformers and GNNs 46

4.1 The Architecture: Transformer ................................. 48
4.2 The Architecture: GNN ........................................... 51
4.3 Numerical Experiments ........................................... 53

5 Device Modeling: GANs 59

5.1 The Dataset ...................................................... 62
5.2 The Architecture .................................................. 64
5.3 Numerical Experiments ........................................... 66

6 Deep-Fake Signals: GANs 70

6.1 The Dataset ...................................................... 71
6.2 Numerical Experiments ........................................... 73

7 Conclusions 80
Contents

Appendices 81

A Derivations: ANNs 82

B Derivations: FDTD 93

References 104
# List of Figures

1.1 Typical AI tasks .................................................. 3

2.1 Operations of the FFNs ........................................... 9

2.2 Operations of the Convolutional layer .......................... 11

2.3 Operations of the Recurrent layer ............................... 13

2.4 Non-linear activation functions ................................. 15

2.5 Typical TensorFlow 2 code ....................................... 18

2.6 Typical TensorFlow 2 code (continued) ......................... 19

2.7 Typical TensorFlow 2 code (continued) ......................... 20

3.1 Yee cell and Yee grid ............................................ 29

3.2 2D TE\(^z\) Yee grid and update equations ...................... 30

3.3 2D TE\(^z\) propagation in the Yee grid: cavities .............. 31

3.4 2D TE\(^z\) split-PML absorbing BC ............................ 32

3.5 2D TE\(^z\) propagation in the Yee grid: Free space .......... 33

3.6 2D TE\(^z\) TF/SF excitation ...................................... 34
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7</td>
<td>2D TE(^z) propagation in the Yee grid: TF/SF excitation</td>
<td>35</td>
</tr>
<tr>
<td>3.8</td>
<td>The ERD architecture</td>
<td>38</td>
</tr>
<tr>
<td>3.9</td>
<td>Numerical experiments with the ERD model</td>
<td>40</td>
</tr>
<tr>
<td>3.10</td>
<td>DDM-based parallelization using the ERD model</td>
<td>43</td>
</tr>
<tr>
<td>4.1</td>
<td>The Transformer architecture</td>
<td>49</td>
</tr>
<tr>
<td>4.2</td>
<td>The CGNN architecture</td>
<td>52</td>
</tr>
<tr>
<td>4.3</td>
<td>Transformer’s predictions in the Continuous Prediction mode</td>
<td>54</td>
</tr>
<tr>
<td>4.4</td>
<td>CGNN’s predictions in the Test Prediction mode</td>
<td>55</td>
</tr>
<tr>
<td>5.1</td>
<td>The LPFDA dataset</td>
<td>63</td>
</tr>
<tr>
<td>5.2</td>
<td>The GAN architecture</td>
<td>65</td>
</tr>
<tr>
<td>5.3</td>
<td>Validation of GAN implementations using the MNIST dataset</td>
<td>68</td>
</tr>
<tr>
<td>5.4</td>
<td>Training GANs on the LPFDA datasets</td>
<td>69</td>
</tr>
<tr>
<td>6.1</td>
<td>Emissions dataset and STFT</td>
<td>72</td>
</tr>
<tr>
<td>6.2</td>
<td>Baselines for membership in a category</td>
<td>74</td>
</tr>
<tr>
<td>6.3</td>
<td>New signals by rearranging spectra</td>
<td>75</td>
</tr>
<tr>
<td>6.4</td>
<td>Similarity of the new signals with the dataset</td>
<td>76</td>
</tr>
<tr>
<td>6.5</td>
<td>Samples from the GAN-generated signals</td>
<td>77</td>
</tr>
<tr>
<td>6.6</td>
<td>Similarity of the GAN-generated signals with the dataset</td>
<td>78</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>Back-Propagation for a three-layered FFN</td>
<td>83</td>
</tr>
<tr>
<td>A.2</td>
<td>Derivation of The He and Glorot initialization schemes</td>
<td>86</td>
</tr>
<tr>
<td>B.1</td>
<td>Derivation of the 2D TE$^z$ update equations</td>
<td>94</td>
</tr>
<tr>
<td>B.2</td>
<td>Derivation of the split-PML absorbing Boundary Condition</td>
<td>98</td>
</tr>
<tr>
<td>B.3</td>
<td>Derivation of the TF/SF consistency updates</td>
<td>101</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Nowadays an increasing volume of data is being collected, from a greater range of sources, and at greater speeds than ever before. These vast collections of data, referred to as 'big data', are potentially a rich resource for the digital economy and extracting valuable information from them can result in considerable benefits. Machine Learning (ML) is the key technology in realizing the economic and social benefits expected from big data. It could lead to increased productivity by enhancing our capability to extract insights from ever-increasing volumes of data. Some industries such as online retailers, social networks, and virtual personal assistants, are already creating value from its use. Other industries can also gain a wide range of benefits that arise from ML’s potential to streamline, improve and/or transform the existing processes. Pharmaceuticals, energy infrastructure, manufacturing, transportation (autonomous vehicles) and healthcare are just a handful of industries that are witnessing emergent applications of ML. In scientific research, ML could become a key tool for driving operational efficiencies as researchers utilize it, in a growing number of ways, to detect previously unforeseen patterns in, or to extract unexpected insights from, data. An example is detecting new fundamental particles in the collision data gathered by particle accelerators. Another example is producing
Chapter 1. Introduction

a fine-grain, local level picture of the impacts of climate change using weather data gathered from many regions to enhance the low-resolution, global-level predictions, created by conventional computational models [1, 2, 3, 4].

While the most prominent use cases of ML in scientific research, as discussed above, have been to identify patterns in collected data that would not have been possible otherwise, little research has been done on the possibility and potential benefits of using measurement and/or simulation data to characterize physical processes. This work examines the possibility of leveraging data to construct ML-based, data-driven models that can emulate physical processes. The outcome of this endeavor will be valuable in developing a roadmap for realization of the transformative potential of ML in computational and experimental modes of scientific investigation.

1.1 Artificial Intelligence

Artificial Intelligence (AI) refers to computer programs that process linguistic and visual information to perceive and interact with the world like humans [11]. Algorithms that process linguistic information (textual and auditory) are categorized as Natural Language Processing (NLP) and the ones that process visual information are categorized as Computer Vision (CV). Examples from typical AI tasks are shown in Fig 1.1.

In the NLP category, Speech Recognition and Text-to-Speech Synthesis refer to transforming spoken sound to text and transforming text to sound respectively. Digital Personal Assistants, shown in Fig 1.1a, combine both of these tasks to interact with humans. Machine Translation refers to transforming text in one language into another. The translation system shown in Fig 1.1b can translate text from more than one hundred languages.

In the CV category, Object Detection refers to localizing and classifying objects in an
Chapter 1. Introduction

Figure 1.1: Typical AI tasks. For image credits see [5, 6, 7, 8, 9, 10].

Figure 1.1: Typical AI tasks. For image credits see [5, 6, 7, 8, 9, 10].
image. The autonomous driving system shown in Fig 1.1c finds surrounding vehicles by localizing (determining pixel locations using a bounding box) and classifying objects in images captured by a front camera and then provides a distance estimate for each. *Pose Estimation* refers to detecting the position and orientation of objects or humans in a scene. The robotic arm shown in Fig 1.1d predicts 3D position and orientation of rigid objects for the purpose of manipulating them (picking, moving, and placing) in interactions with human collaborators. *Image Synthesis* refers to generating a modified version of an existing image or generating an entirely new one. The facial expression generation system shown in Fig 1.1e generates various emotional expressions from a neutral face while preserving its identity. *Image Transcription* refers to observing unstructured visual representation of some type of data and transcribing the information into discrete textual form. Image captioning system shown in Fig 1.1f observes the scene depicted in the figure and provides a textual statement about its content. *Image Super-Resolution* refers to generating a higher resolution version of the original image. Image enhancement system shown in Fig 1.1g enables fast acquisition of high-resolution MRI images by producing high-resolution reconstructions from low resolution acquisitions. And finally, *Image Denoising* refers to recovering the original image signal from its corrupted form. Image restoration system shown in Fig 1.1h reconstructs MRI scans from under-sampled inputs.

### 1.2 Machine Learning

Machine Learning (ML) refers to computer programs that accomplish their tasks by *learning* from examples. A computer program is said to learn from experience $E$ with respect to some class of tasks $T$ and performance measure $P$, if its performance at tasks in $T$, as measured by $P$, improves with experience $E$ [12]. The *experience* that is available to the program is represented by the *dataset* which itself is a collec-
Chapter 1. Introduction

tion of examples. An example consists of a pair of datapoint and its corresponding label/target. A datapoint, is a collection of features that have been measured from some object or event. For classification tasks the output is discreet-valued while for regression tasks the output is continuous-valued. Therefore, the performance measure is task specific. For classification tasks the accuracy (i.e., the proportion of examples for which the program produces the correct outputs) is used as the performance measure while for regression tasks the expected value of the error is used.

The ML model, denoted by $f(x, \theta)$, must perform its task in the inference phase with minimal risk. The risk (also referred to as the generalization error), denoted by $R(\theta)$, is defined as the expected value of the error (i.e., the output of the loss function $L$) for examples that the model encounters in the inference phase (these examples are thought of as drawn from a data-generating distribution $p_{\text{data}}$). Therefore, $R(\theta) = \mathbb{E}_{(x, y) \sim p_{\text{data}}} [L(f(x, \theta), y)]$. However, since it is impossible to access all the possible examples for the model’s task, the learning algorithm must minimize the risk indirectly. It attempts this by minimizing the empirical risk on a training set of examples hoping that doing so will also reduce the risk. The empirical risk, denoted by $R_{\text{emp}}(\theta)$, is defined as the expected value of the error for examples in the training set (these examples are thought of as drawn from a data-generating distribution $\hat{p}_{\text{data}}$). Therefore, $R_{\text{emp}}(\theta) = \mathbb{E}_{(x, y) \sim \hat{p}_{\text{data}}} [L(f(x, \theta), y)]$.

In the training phase, the learning algorithm tunes the model by processing a training set of examples, computing the cost, and then adjusting the model’s parameters so that the cost is minimized. In other words, the learning algorithm tunes the model by adjusting its parameters such that $\frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}, \theta), y^{(i)})$ is minimized. Once the training is finished, the model enters the inference phase in which it will process previously unseen examples. If the model exhibits minimal cost and also a minimal gap between the cost and the generalization error, it is said that it generalizes well. On the other hand, if the cost is minimal but there is a large gap between the cost and the generalization error, it is said that the model overfits the training set.
Computation is recognized as an important tool for research in many of areas of science in general, and physics in particular [13]. Computational Physics (CP) combines the quantitative theories of physics with numerical analysis and computer programming to produce numerical simulations of physical processes. The quantitative theories of physics result in equation-based models, i.e., Partial Differential Equations (PDEs) and Ordinary Differential Equations (ODEs), that precisely describe the spatial distribution and temporal evolution of a physical quantity given some boundary and initial values. In CP, numerical analysis is used to adapt these equation-based models to discretized schemes that can be computed using a computer. CP complements the experimental branch of scientific investigation when, due to the complexities of a theory, e.g., Quantum Chromodynamics, numerical simulation is the only possible method of obtaining predictions to test against observations, or when controlled experiments are not possible at all, e.g., when astronomers study the development of large-scale structures in the early universe, the evolution of stars or collision of galaxies. CP also plays a crucial role in advancements of technology. Numerical simulations inform initial design concepts and are indispensable in design development and optimization as they significantly lessen the amount of physical testing that must be carried out to validate a design and measure its performance.

An example is the use of Computational Fluid Dynamics (CFD) by mechanical engineers to compute the drag, lift, structural and thermal loads on an aircraft in the process of developing optimal airframes. Another example is the use of Computational Electromagnetics (CEM) by electrical engineers to compute the radiation pattern of antennas and scattering parameters of RF circuits as they develop optimal components for communication systems.

In recent years, two major developments have opened new perspectives for developing none-equation-based (equation-free), data-driven models for simulating physical
Chapter 1. Introduction

processes. The first, is the rapid advancements in the field of Deep Learning which have produced learning algorithms with unprecedented efficacy. The second, is the ever-increasing computing power and greater availability of data storage which have enabled easy generation and storage of large amounts of data for training Deep Learning models.

The aim of this research is to introduce new data-driven methodologies for CEM and for antenna design and development. These two areas of interest form the two thrusts of this research activity. In Thrust-I, efforts are motivated by the observation that an ML model can act as a fast surrogate for its equation-based counterpart, and the goal is to leverage the readily available simulation data to characterize the physical processes involved in propagation, reflection and scattering of electromagnetic waves using suitable state-of-the-art models. A potential application for the methodology developed in this thrust is augmenting and further accelerating the fast electromagnetic simulators. In Thrust-II, efforts are motivated by the observation that generative ML models can produce new examples that are similar to the ones that they were trained with, and the goal is to leverage the readily available simulation data to characterize the key features of favorable and near-optimal designs in application-specific scenarios using state-of-the-art generative models. A potential application for the methodology developed in this thrust, is augmenting antenna design knowledge management systems.

1.4 Reproducibility

The code, data and outputs for all the experiments reported in this dissertation are available in dedicated repositories. These, and much more useful information, can be easily accessed through the following GitHub page which is dedicated to this dissertation: github.com/oameed/unm_phd_dissertation.
Chapter 2

Deep Learning

Artificial Neural Networks (ANNs) constitute a category of ML models. Inspired by the biological brain, they are computational models that are composed of multiple processing layers intended to learn representations of data at multiple levels of abstraction. In recent years, adoption and use of ANNs has considerably improved the state-of-the-art in various AI tasks [12, 14].

2.1 Feed-Forward Networks

A Feed-Forward Network (FFN) is shown in Fig. 2.1a. It consists of a stack of layers. The first layer accepts the input vector, denoted by $a^{(0)}$ (since it can be thought of as the output of the zeroth layer), and is referred to as the input layer. The last layer produces the output vector, denoted by $a^{(3)}$, and is referred to as the output layer. All the layers in between these two layers are referred to as the hidden layers. The input to each layer is the output of its previous layer; For example, the input to the second layer is $a^{(1)}$ which is the output from the first layer.
Chapter 2. Deep Learning

Figure 2.1: (a) forward pass and back propagation (b) single unit

Each layer consists of several units (also referred to as neurons) that independently process that layer’s input vector. The vector containing the outputs from all the units in that layer constitutes its output vector. A typical unit is shown in Fig. 2.1b. It produces its output by multiplying each element of the input vector by a dedicated weight, summing them together along with a bias and, finally, applying a non-linear function to the total sum. Since each unit processes all the elements of the input (by multiplying each and every element by a dedicated weight), this type of layer is referred to as a fully-connected layer. For any layer $l$, all the weights are packed into a weight matrix, denoted by $W^{(l)}$, such that each row contains all the weights of a single unit. The weight matrix is expanded to also include the biases by inserting
Chapter 2. Deep Learning

the bias of each unit at the end of its corresponding row. The resulting expanded matrix is denoted by $\tilde{W}^{(l)}$. Correspondingly, the input vector $a^{(l-1)}$ is expanded by inserting a one at its end and the resulting expanded vector is denoted by $\tilde{a}^{(l-1)}$.

Given the expanded weight matrix and the expanded input vector, the output of the $l^{th}$ layer is computed by $a^{(l)} = f(\tilde{W}^{(l)} \tilde{a}^{(l-1)})$ which is referred to as the forward propagation of the input.

In the training phase, the learning algorithm requires the gradients of the cost with respect to all the weights and biases in each layer so that it can adjust those weights and biases such that the cost is minimized. As is demonstrated in Fig. 2.1a, the gradients for the $l^{th}$ layer, denoted by $\frac{\partial E}{\partial \tilde{W}^{(l)}}$, are computed using $\delta^{(l)}$ which is a vector containing the partial derivatives of the error with respect to each element of the output vector of that layer. As is derived in Table A.1, once $\delta$ is computed at the output of the last layer, the $\delta$ at the output of all the remaining layers are computed by $\delta^{(l)} = W^{(l+1)T} \left( f^{(l+1)} \odot \delta^{(l+1)} \right)$ which is referred to as the back propagation of the error.

2.2 Convolutional Layer

A convolutional layer is a layer with two properties, namely, sparse interactions and weight sharing. The first property refers to the fact that each element of the output is produced by the interactions of the layer’s set of weights (which are much smaller in size compared to the input) with only a local region of the input. The second property refers to the fact that the same set of weights interacts with different local regions of the input to produce their corresponding elements of the output. As such, the convolutional layer stands in contrast to the fully-connected layer in which a unit’s output is produced by the interactions of separate weights with every element of the input [12]. The weight sharing causes the layer’s output to be equivariant to
Chapter 2. Deep Learning

Figure 2.2: (a) and (b) Typical Convolutional layers (c) Transpose Convolutional layer

**translation.** This means that, in the context of processing images, if a certain feature moves across the input image, its corresponding output representation will also move in the same way. Similarly, in the context of processing time series data, if an event is advanced (or delayed) in time, its output representation will also be advanced (or delayed).

These two properties of the convolutional layer, namely, its sparse interactions with
its input and its output’s equivariance to translations, specialize it to the type of input
for which repeated application of a function (which maps a small neighborhood of
input elements to some representation) to multiple positions across that input will
be useful. An example would be detecting edges across an image as part of an object
detection task. It would be useful to apply a convolutional layer to the input given
that each edge constitutes only a small portion of the input and that they can appear
at any position in the input.

A convolutional layer is shown in Fig. 2.2a. It produces the elements of the output
by sliding its set of weights, referred to as the filter, across the the input. For each
position, the weights in each channel of the filter are multiplied by their corresponding
elements in the input (i.e., same channel and same position in the input) and the
results are summed together. The sums from all the channels are added together
along with a bias. Finally, a non-linear function is applied to the total sum.

The size of the output, in each dimension, depends on the size of the input and the
size of the filter and can be controlled by padding, stride and dilation. Padding refers
to insertion of zeros to either side (beginning or end) of the input in each dimension.
Stride refers to the distance between two consecutive positions of the filter. Dilation
refers to the inflation of the filter by inserting spaces between filter elements; This
provides a convenient way to increase the receptive field (i.e., the local region of
interaction) of the filter without increasing its size.

A transpose-convolution layer, refers to a convolutional layer that produces an output
which is larger in size than its input. One way to achieve this, as shown in Fig. 2.2c,
is to pad the input such that the operation of a convolutional layer, with \( S = 1 \), on
the padded input will result in the desired output size [15].
Chapter 2. Deep Learning

2.3 Recurrent Layer

A recurrent layer is a layer which is specialized for processing inputs that are in the form of sequences. This type of layer can learn a single model for processing all the positions of an input sequence. This capability results from weight sharing across all the positions of the input sequence which is achieved through recurrent updates. A recurrent update refers to the application of the same forward propagation operation (i.e., multiplication by the same set of weights) to produce the current position in the output from the current position in the input and the previous position in the output.

An example use case for the recurrent layer would be predicting the next word in a sentence as part of a language translation task. The single shared model that it learns represents the underlying linguistic rules for all positions of the input sentence. In contrast, a fully-connected layer will have a separate set of weights for each position of the input sequence and, therefore, it will need to learn the rules for each position separately.

Figure 2.3: (a). Vanilla recurrent layer (b). LSTM
Chapter 2. Deep Learning

2.4 Deep FFNs

The *Universal Approximation Theorem* (UAT) states that a FFN with a linear output layer and at least one hidden layer which has any non-linear activation function (i.e., either a ReLU or a 'squashing' function such as the sigmoid) can approximate any Borel-measurable function (i.e., a continuous function defined on a closed and bounded subset of $\mathbb{R}^n$) with any desired non-zero amount of error, provided that the FFN is given enough hidden units. While the UAT guarantees the existence of the function that would generalize to unseen examples, it provides no guarantees that the learning algorithm will be able to learn that function. Specifically, learning can fail either due to the optimization algorithm’s inability to find the set of parameters that corresponds to the desired function or due to overfitting. Furthermore, while the UAT guarantees the existence of a FFN wide enough to achieve any desired degree of accuracy, it provides no indication as to how wide it should be. In the worst case, an exponential number of hidden units may be required. Therefore, in summary, a FFN with a single hidden layer is sufficient to represent any function, but the layer may fail to learn at all or to generalize and it may be infeasibly large. Structuring FFNs as deep stacks of layers reduces the number of units required to represent the desired function and reduces the generalization error. Furthermore, deep FFNs convey the point of view that the sought after function should be constructed by composition of several simpler ones.

When training a deep FFN, a major obstacle for the learning algorithm is the failure of the back-propagation procedure. When the gradients of the error w.r.t. the weights and biases of the $l^{th}$ layer, i.e., $\frac{\partial E}{\partial W(l)}$, become so small that they can't effect meaningful change in that layer’s parameters, the learning algorithm is said to suffer from *vanishing gradients*. Likewise, when $\frac{\partial E}{\partial W(l)}$ become so large that they cause unstable change, the learning algorithm is said to suffer from *exploding gradients*. The problems of vanishing/exploding gradients at the start of training can be mitigated...
by adoption of non-saturating non-linear functions and by proper initialization of layer weights together with standardization of the input.

Figure 2.4: Non-Linear Activation Functions

The non-linear function of choice for activation of all layers is the ReLU. As shown in Fig. 2.4(b), its derivative for positive values (and, by definition, at zero) is one and for negative values is zero. In computation of $\delta$ for deeper (i.e., lower) layers, the derivative of the non-linear function from all the previous layers are multiplied together. Since for the ReLU $f'(z \geq 0) = 1$, these multiplications will not attenuate the elements of the $\delta$. While $f'(z < 0) = 0$ does contribute to zero elements in $\delta$ and, consequently, in $\frac{\partial E}{\partial W}$, the back-propagation procedure will work as long as the gradients can propagate along some paths; In other words, as long as $z \geq 0$ for some of the units in each layer [16]. In contrast to the ReLU, both the hyperbolic tangent and the sigmoid functions cause significant attenuation in elements of the $\delta$ in deeper layers even when their inputs are from a small interval centered at zero (i.e., are not from the saturated regions). For the hyperbolic tangent, as shown in Fig. 2.4(b), the maximum of the derivative is 1 and, therefore, multiplication of many of these values will result in a significant attenuation. For the sigmoid function, the maximum of
Chapter 2. Deep Learning

the derivative is 0.25 which results in an even more severe attenuation. The gradual attenuation of the gradients with increasing depth is not the only mechanism by which the sigmoid function can cause vanishing gradients. If used at the output (i.e., the top) layer, it can cause the $\delta$ to attenuate to near zero at the top and effectively vanish for the lower layers. The reason for this is that, at the start of training, the transformation that is computed by the lower layers is not useful and, therefore, the output of the last layer relies more on that layer’s biases than its previous layer’s output (which is ultimately derived from the input). As such, initially, the gradients would modify the weights of the previous layer such that its output is close to zero which corresponds to the saturation region of the sigmoid [17].

Proper weight initialization ensures that, at the start of training, the magnitudes of $a^{(l)}$ and $\delta^{(l)}$ will not attenuate to zero or grow very large. With proper initialization, in the forward pass, starting from an input with zero mean and unit variance, the variance of the elements of the output of each layer remains the same as that for the layer below it. Likewise, in the back-propagation, the variance of the elements of the $\delta$ at the output of each layer remains the same as that for the layer above it. The proper initial variance for layer weights are derived in Table A.2 [17, 18, 19, 20].

For layers with ReLU activation, the weights can be drawn from either a normal distribution $w^{(l)} \sim \mathcal{N}(0, \frac{2}{O_{l-1}})$, in which $O_{l-1}$ denotes the size of the input to the $l^{th}$ layer, or a uniform distribution $w^{(l)} \sim U[-\sqrt{\frac{6}{O_{l-1}}}, \sqrt{\frac{6}{O_{l-1}}}]$. This is referred to as the He initialization. For layers with hyperbolic tangent activation, the weights can be drawn from either a normal distribution $w^{(l)} \sim \mathcal{N}(0, \frac{2}{O_{l-1}})$ or a uniform distribution $w^{(l)} \sim U[-\sqrt{\frac{6}{O_{l-1} + O_{l}}}, \sqrt{\frac{6}{O_{l-1} + O_{l}}}]$. This is referred to as the Glorot initialization.

When training a deep FFN, the learning algorithm can also greatly benefit from learning the moments of distribution of the elements of the input to each layer alongside the weights and biases of that layer. The batch normalization (BN) procedure [21, 22], in the first step, standardizes each element of the layer’s input with the mean and variance that is computed over the current batch for that element
and, in the second step, scales and shifts the standardized elements with learned parameters. In the training phase, for $a_k^{(l-1)}$, the BN procedure computes $\mu_{k,B}$ and $\sigma_{k,B}^2$ from the entire batch and then transforms it to $a_k' = \gamma_k a_k^{(l-1)} + \beta_k$ in which $a_k' = \frac{1}{\sigma_{k,B}}(a_k^{(l-1)} - \mu_{k,B})$. In the inference phase, $a_k^{(l-1)}$ is computed using the average of the values computed for $\mu_{k,B}$ and $\sigma_{k,B}^2$ during the training.

### 2.5 Deep Learning Frameworks

TensorFlow [23] and PyTorch [24] are two open-source deep learning frameworks. TensorFlow was developed by the Google Brain team as the successor to the DistBelief framework (Google’s internal deep learning infrastructure in operation since 2011) and was open-sourced and released in 2015 [25]. Google released TensorFlow 2 in 2019 which included many improvements such as tight integration of the Keras library [26]. PyTorch was developed at the Facebook’s AI Research (FAIR) lab as the successor to the Torch framework (a scientific computing library, based on the Lua programming language, with a focus on machine learning that existed since 2002) and was released in 2018 [27].

The experiments reported in this work use the TensorFlow framework. The code structure is shown in Fig. 2.5 to Fig. 2.7. Each experiment involves the execution of two separate scripts, first the training and then the inference. The training script defines the model, configures the utilities for real-time monitoring, checkpointing and history, prepares and loads the training set, and finally, runs the training session. While in the training session, the model is checkpointed at specific intervals. Once the training is finished, the inference script loads the trained model from the last checkpoint, loads the test set and makes predictions by applying the model to the data.
Chapter 2. Deep Learning

Figure 2.5: Typical TensorFlow 2 code

```python
import tensorflow as tf

# Training
seq_network_1 = get_seq_network_1(arg_1, ...)
model = ml_algorithm(seq_network_1=seq_network_1, ...)
model.compile(optimizer=tf.keras.optimizers.Adam(...),
              loss=...,
              metrics=...)
callbacks=[callback_custom_checkpoint(),
            callback_custom_monitor(...),
            callback_custom_history(...),
            tf.keras.callbacks.Tensorboard(...)]
save_data_to_tfrecords_format(arg_1,...)
data_train = read_tfrecords(mode='train',...)
data_validation = read_tfrecords(mode='validation',...)
model.fit(x=data_train,
          validation_data=data_validation,
          callbacks=callbacks,
          ...)

# Inference
import tensorflow as tf
model = tf.keras.models.load_model(path_to_checkpoint,
                                    ...)
data = ...
predictions = model(data)
...
```

Figure 2.5: Typical TensorFlow 2 code
import tensorflow as tf

def get_seq_network(arg_1,...):
X = tf.keras.Input(shape=..., ...)

x = tf.keras.layers.Dense (...)(X)
x = tf.keras.layers.BatchNormalization(...)(x)
x = tf.keras.layers.ReLU (...)(x)

return tf.keras.Model(inputs=X, outputs=x, name=...)

class ml_algorithm(tf.keras.Model):
    def __init__(self, seq_network_1,...):
        super().__init__()
        self.network_1=seq_network_1

        def compile(self, optimizer, loss,<metrics>):
            super().compile()
            self.optimizer=optimizer
            self.loss =loss
            self.<metrics>...</metrics>

def call(self,X,training=None):
    x=self.<network>(X,training=training)

    return ...

    def train_step(self,data):
        if isinstance(data,tuple):
            x,y=data
            ...
        with tf.GradientTape() as tape:
            predictions=self(x,training=True)
            ...
            loss =self.loss(y,predictions)
            gradients =tape.gradient ( loss , self.<network>.trainable_weights )
            self.optimizer.apply_gradients(zip(gradients, self.<network>.trainable_weights))
            return ('loss': loss, ...

class callback_custom_checkpoint(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        tf.keras.models.save_model(self.model ,path_to_checkpoint)

Figure 2.6: Typical TensorFlow 2 code (continued). Details for segments one to three.
import tensorflow as tf
...

def save_data_to_tfrecords_format(...):

def serialize_example(x, y):
  feature = ('x': tf.train.Feature(bytes_list=tf.train.BytesList(value=[x.tostring()])),
             'y': tf.train.Feature(int64_list=tf.train.Int64List(value=[y]))
  example_proto = tf.train.Example(features=tf.train.Features(feature=feature))
  return example_proto.SerializeToString()

def write_serialized_example(x, y, node):
  filename = ...
  with tf.io.TFRecordWriter(filename) as writer:
    for i in range(number_of_examples):
      example = serialize_example(x[i], y[i])
      writer.write(example)

def get_train_validation_splits(x, y, ...):

  return x_train, y_train, x_validation, y_validation

x, y = get_data(...)
x_train, y_train, x_validation, y_validation = get_train_validation_splits(x, y,...)
write_serialized_example(x_train, y_train, 'train')
write_serialized_example(x_validation, y_validation, 'validation')

import tensorflow as tf
...

def read_tfrecords(node, ...):
    ...

  feature = ('x': tf.io.FixedLenFeature([], tf.string),
             'y': tf.io.FixedLenFeature([], tf.int64))

  def parse_function(example_proto):
      parsed_example = tf.io.parse_single_example(example_proto, feature)
      x = tf.io.decode_raw(parsed_example['x'], tf.float32)
      y = parsed_example['y']
      ...
      return x, y

dataset = tf.data.TFRecordDataset(filenames)
dataset = dataset.map(parse_function)
dataset = dataset.batch(batch_size, drop_remainder=True)
dataset = dataset.shuffle(buffer_capacity, reshuffle_each_iteration=True)
return dataset

Figure 2.7: Typical TensorFlow 2 code (continued). Details for segments four and five.
Chapter 3

Emulation of the FDTD: RNNs

Electromagnetic phenomena are mathematically modeled using Maxwell’s equations which, in their differential form, comprise a system of four, first-order Partial Differential Equations (PDEs). In a region of three-dimensional space, these equations relate the curl (circulation) of the electric and magnetic fields to their temporal rate of change and to the electric and magnetic current densities. They also relate the divergence (outpouring) of the electric and magnetic fields to the distribution of the electric and magnetic charge densities. These equations provide a quantitative description for both radiation (i.e., excitation of electromagnetic fields by electric and magnetic currents) and propagation phenomena. In the study of radiation, an important problem is finding the distribution of electric and/or magnetic current densities that are excited on the surface of an object due to a uniform wavefront impinging on it. This is called the scattering problem and is formulated in terms of an Integral Equation (IE) that has the desired unknown currents both inside and outside of the integral operator and the impinging wave as its forcing term. In the study of propagation, an important problem is finding the field configurations (modes) that are supported by the environment that the wave is propagating in (free space, layered media, guiding structures) using electric and/or magnetic vector potentials. This is
called the eigen-mode problem and is formulated in terms of a second-order, homogeneous PDE for the desired unknown vector potential. While closed-form solutions for the PDEs of the eigen-mode problem exist only for the most simplistic of geometries (rectangular and circular), the IEs of the scattering problem don’t admit a closed-form solution at all. Therefore, numerical methods are indispensable tools for studying the solutions of IEs and PDEs in radiation and propagation problems. The Method of Moments (MoM) and the Finite Element Method (FEM) are the numerical methods of choice for IEs and PDEs respectively.

In MoM [28, 29, 30], the domain of the problem is divided into non-overlapping segments and a basis function is associated with each segment such that its value is non-zero over that segment and is zero at any other point of the domain. In the simplest implementation, the basis functions have a constant, but unknown, value over their associated segment. In this scheme, the magnitude of the unknown function (current density) is approximated by the sum of all the basis functions. This, results in transformation of the original problem (i.e., integration of an expression involving an unknown function over the entire domain) into integration of an expression involving a constant multiplier over each segment. To determine the values of the basis functions, the weighted-residual method is used to construct a system of equations. In the simplest implementation, the weighing functions are impulses, each of which is shifted from the origin to the center of a segment. Using a shifted impulse as the weighing function results in fixing the observation point, in the sum of integrations, to the location of that impulse. Therefore, each equation, in the system of equations, can be constructed by carrying out the integrations over each segment while the observation point is fixed to the center of a particular segment. The solution to this system fully determines the basis functions associated with each segment and therefore represents an approximation to the solution of the IE over the entire domain of the problem.

In FEM [31, 32, 33], the domain of the problem is divided into non-overlapping seg-
Chapter 3. Emulation of the FDTD: RNNs

ments, called finite elements. Over each finite element, the unknown function (a component of the electric or magnetic vector potential) is interpolated using a set of polynomial basis functions that are defined by the values of the unknown function at the nodes of the element. To determine the values of the unknown function at the nodes of each element, the Galerkin weighted-residual method is used to construct a system of equations. In this method, the weighing functions for each element are the same as its basis functions. Furthermore, over each finite element, the Galerkin weighted-residual method results in transformation of the second-order PDE into an integral equation that has both the unknown function and the weighing function under the integral operator. In this scheme, the integration of the unknown function over the finite element is transformed into the sum of the integrations of the interpolating functions over that element. This, results in a system of equations for the unknown node values of that element. To determine the node values over all the finite elements, a global system of equations for node values is assembled by adding the contributions from all finite elements to each node. The solution to this global system fully determines the interpolating functions over all elements and therefore represents an approximation to the solution of the PDE over the entire domain of the problem.

In contrast to both MoM and FEM, which compute the spatial distribution of time-harmonic quantities (e.g., magnitude of current densities or vector potentials) by constructing a system of linear equations, the Finite Difference Time Domain (FDTD) is a time-marching scheme for electric and magnetic fields in a region of space. In FDTD [34, 35, 36], the first-order, time-domain PDEs of Maxwell’s system (specifically, the two curl equations) are discretized in spatial dimensions using the Yee grid and are forwarded in time using the interleaved leapfrog procedure. In each cell of the Yee grid, the electric field components are placed at the centers of the edges and are oriented parallel to them. The magnetic field components are placed at the centers of the faces and are oriented normal to them. When all neighboring cells are
considered, each electric field component is surrounded by four magnetic field components and vice versa. This arrangement of field components in each cell, enables the discretization of each component of the curl equations using the second-order-accurate central-difference scheme. The resulting update equations connect electric and magnetic field components across the neighboring cells, and from one time-step to the next, in such a way that Maxwell’s equations are satisfied. Therefore, the variations of the field components in a region of space that are obtained using this set of update equations represent electromagnetic waves propagating through that region. In this method, to obtain any quantity of interest at a particular cell, or a neighborhood of cells, the grid is excited at a specific location and then enough time is allowed to pass so that the propagating wave can reach that region. As the wave passes through, all the field components are recorded and used for further calculations.

In FDTD, the propagation of electromagnetic waves in a region of space are modeled using the combination of the Yee grid and a set of update equations. The Yee grid defines a region of space as an aggregate of cells, each of which has an electric and a magnetic field vector associated with it. The update equations are the dynamics that enable the local disturbances in field magnitudes to propagate through the entire grid as electromagnetic waves. Any cross-section of the Yee grid that is parallel to one of the three spatial axes will contain three field components, two that are parallel and one that is perpendicular to it. Therefore, the cross-section will contain three grids of values that change over time as effected by the dynamics of electromagnetic phenomena. This arrangement exhibits striking parallels with a sequence of images, i.e., a video. Each grid of field values that lie on the cross-section can be thought of as a grid of pixels or, in other words, a single-channel (monochrome) image. Therefore, the three grids of field values that lie on the cross-section constitute a three-channel (color) image. Then, the temporal variations of the grid values constitute a sequence of color images or, in other words, a video. This interpretation connects the domain
Chapter 3. Emulation of the FDTD: RNNs

of Computational EM (CEM) with the domain of Computer Vision (CV) and recasts the FDTD algorithm for time-marching of EM fields as the task of predicting the future frames of a video.

With the aim of constructing a data-driven and learned model of EM dynamics, we investigate whether FDTD’s set of update equations can be substituted with an RNN. In other words, whether an RNN can produce a sequence of grid values, i.e., a video, that evolve as electromagnetic waves. As examples, we consider the problem of plane wave propagation and plane wave scattering from perfect electric conductors (PECs) and set up experiments in which an RNN is first trained with a set of FDTD videos and then, in inference phase, is presented with an FDTD video that it has not seen during training and is asked to produce its next few frames [37].

The dataset for the experiments comprises videos that were created by splitting several complete simulations into shorter sequences. An in-house developed FDTD solver was used to simulate various scenarios of propagation and scattering as the angle of incidence for TF/SF excitation, locations of point source excitations and the shape, size and locations of the objects were varied randomly.

The model architecture for the experiments comprises a convolutional encoder, a convolutional LSTM and a convolutional decoder and is implemented using TensorFlow. When a video is presented to the model, the convolutional encoder extracts its first frame (or alternatively, its last frame if operating in the Continuous Prediction mode in the inference phase) and reduces that frame to a set of latent space features that are then fed to the LSTM. With the extracted features as initial input, the hidden state of the LSTM is updated for a specific number of times to produce a stack of features that can be interpreted as future frames in the latent space. Finally, the stack of features is fed to the decoder to construct complete future frames of the EM fields for that video. In inference phase, this Encoder- Recurrent-Decoder (ERD) architecture can generate many frames into the future by feeding the generated video back as input. Using the Frobenius norm of the difference between the
Chapter 3. Emulation of the FDTD: RNNs

ground-truth frame and the predicted frame as a measure of accuracy, it is observed that this quantity remains within 5% of its maximum value over most of the generated frames. Furthermore, the run time of the model for generating a single frame exhibits $17 \times$ speedup compared to the run time of a single update of the FDTD solver. These experiments successfully demonstrate that the ERD architecture can be utilized as a data-driven and learned model to emulate EM dynamics.

With this experimental affirmation, we proceed to explore the realm of possible applications for our data-driven and learned model of EM dynamics. We propose to use this model as a fast computation tool in a Domain Decomposition Method (DDM) based parallelization scheme. As the computational domain grows larger, the run time of computations increases rapidly. One approach to mitigating this rapid increase is DDM-based parallelization of computations. In this scheme, the entire computational domain is divided into smaller regions, called sub-domains, and computations are carried out in parallel over each. Furthermore, each sub-domain exchanges boundary value information with its neighbors to preserve the global structure of the original physical problem. Since events that occur in each sub-domain are similar to the ones that occur in others, e.g., a wavefront propagates from one corner to another or a wavefront impinges on an object and scatters, a learned model of EM dynamics can be effectively put to use for generating the future states of all sub-domains. Also, the large speedups that can be achieved using the learned model, as demonstrated in previous experiments, will further expedite the computations.

3.1 Finite Difference Time Domain

The FDTD method, as mentioned above, is a time-marching scheme for electric and magnetic fields in a region of space. A standalone 3D Yee cell is shown in Fig. 3.1(a). Two 2D field configurations, namely the $\text{TE}_z$ mode and the $\text{TM}_z$ mode can be ob-
tained from it using cut-planes that are parallel to the $x - y$ plane. As shown in Fig. 3.1(b) and Fig. 3.1(c), once all neighboring cells are considered, both curl equations in the Maxwell’s system of equations are satisfied. In this work, the 2D TE$_z$ mode is considered.

The 2D Yee grid and the corresponding update equations for the TE$_z$ mode are shown in Fig. 3.2(a) and Fig. 3.2(b) respectively. The derivation of the update equations are presented in Table B.1. The valid range of cells for the update procedure is shown in Fig. 3.2(c). For the electric field components this range is from the second cell to the last one in each dimension. For the magnetic field component it is from the first cell to the one before last in each dimension. In this scheme, two edges of the grid behave as PECs and the other two behave as PMCs. However, it is favorable to have all the edges of the grid behave as PECs. To achieve this, the electric field components of the last cell in each dimension are excluded from updates. With all the edges behaving as PEC walls, the entire grid behaves as the interior of a 2D cavity. In other words, an excitation enforced at any location of the grid will propagate and reflect from the PEC walls. Examples are shown in Fig. 3.3.

To simulate wave propagation in free space, some form of Absorbing Boundary Condition (ABC) must be enforced at the edges of the grid. The most important ABC is the Perfect Matched Layer (PML). Configuring the cells in the close vicinity of the grid edges as PML results in a region with reflection-less boundary that severely attenuates all the waves that enter it; For the interior cells (i.e., those cells that are not part of the PML region) this emulates free-space conditions. Construction of the PML region is achieved by introducing anisotropy in the electric and magnetic conductivities of the medium. Fig. 3.4(a) shows the modified Maxwell’s equations along with their general solutions in such media for the TE$_z$ mode. As shown in Fig. 3.4(b), these general solutions can be simplified by enforcing the matching conditions. Furthermore, these simplified solutions will cross the boundaries between the PML and the interior regions without any reflections if the parallel electric conductivities
are set to be equal on both sides. The derivation of these results are presented in Table B.2. The resulting implementation scheme and modifications required for the update of the magnetic field are shown in 3.4(c) and Fig. 3.4(d) respectively. Examples are shown in Fig. 3.5.

When an excitation is enforced at a specific location on the grid, it propagates as a spherical wavefront. Planar wavefronts can be implemented using the Total-Field/Scattered-Field (TF/SF) procedure. As shown in Fig. 3.6(a), a boundary is defined such that inside of it all fields are considered to be total fields and outside of it all fields are considered to be scattered fields. To correct the inconsistencies that this scheme induces in the update equations, extra terms must be added to the update equations that represent the excitation. These consistency updates are shown in Fig. 3.6(b) and their derivation is presented in Table B.3. The excitation is in the form of an auxiliary 1D simulation the values of which determine the value of the corrective terms at any point along the TF/SF boundary. Examples are shown in Fig. 3.7.
Figure 3.1: (a). Yee cell (b). and (c). Yee grid
Figure 3.2: (a) 2D TE\textsuperscript{z} Yee grid (b) 2D TE\textsuperscript{z} update equations (c). Range of updated cells for PEC-backed computational domain
Figure 3.3: (a). Rectangular cavity (b). Circular cavity (c). Ridged rectangular cavity
Chapter 3. Emulation of the FDTD: RNNs

Maxwell’s Equations
\[
E_x = -\frac{1}{\omega S_y} \frac{\partial H_z}{\partial y} \\
E_y = -\frac{1}{\omega S_y} \frac{\partial H_z}{\partial x} \\
E_z = -\frac{1}{\omega S_y} \frac{\partial H_y}{\partial x} - \frac{1}{\omega S_y} \frac{\partial H_x}{\partial y} \\
H_x = H_{ex} + H_{in}
\]

Solutions
\[
\begin{aligned}
E &= E_0 \left( \frac{S_y}{S_x} \sin \theta \hat{a}_x + \frac{S_y}{S_z} \cos \theta \hat{a}_z \right) e^{-jkx} \\
H &= \left( \frac{1}{k} E_0 \hat{a}_z \right) e^{-jkx} \\
k &= \sqrt{(\omega S_y \cos \theta \hat{a}_x + \sqrt{S_y S_z} \sin \theta \hat{a}_z)} \\
Z &= \frac{E_0}{H_0} \\
W &= \frac{S_y}{S_x} \cos \theta \hat{a}_x + \frac{S_y}{S_z} \sin \theta \hat{a}_z
\end{aligned}
\]

Figure 3.4: (a). Solutions to Maxwell’s equations in media with anisotropic conductivities (b). Simplified solutions under the matching condition (c). The split-PML scheme (d). Modified update equations

Matching Condition
\[
\begin{align*}
\sigma_x &= E_x \\
\sigma_y &= E_y \\
\end{align*}
\]

Uniform Plane wave
\[
E = E_0(\sqrt{(\omega \mu \sin \theta) + \omega \varepsilon \sin \phi}) e^{-j(wt + kx - \omega \varepsilon \sin \phi t)}
\]
\[
\begin{align*}
H &= \frac{1}{\nu} a_y \times E \\
\end{align*}
\]

\[
E_x = E_0(-\sin \theta \hat{a}_x + \cos \theta \hat{a}_z) \\
E_y = \cos \theta \hat{a}_x - \sin \theta \hat{a}_y
\]

Reflection & Transmission
\[
\begin{align*}
\sigma_{ex} &= \sigma_{ez} \\
\Gamma &= 0 \\
\sigma_{ey} &= \sigma_{ez} \\
T &= 1
\end{align*}
\]

Electric Conductivity Profile

Left/Bottom
\[
\sigma(x) = \sigma_{max} \left( \frac{a_x - a_b}{a_c - a_b} \right)^P
\]

Right/Top
\[
\sigma(x) = \sigma_{max} \left( \frac{a_x - a_c}{a_b - a_c} \right)^P
\]

\[
\sigma_{max} = \frac{2}{\varepsilon_0} \frac{2\pi\varepsilon_0 C}{\Delta x}
\]

\[
\begin{align*}
\Delta x &= \sqrt{(P + 1)(a_c - a_b)} \\
\Delta y &= \sqrt{2A}
\end{align*}
\]

Modified $H_z$ Update
\[
H_x^{n+\frac{1}{2}}(i, j) = C_{hh}(i, j) H_x^{n-\frac{1}{2}}(i, j) + C_{hxy}(i, j) \left[ E_y^n(i, j + 1) - E_y^n(i, j) \right] \\
C_{hh}(i, j) &= \frac{2\mu(i, j) - \Delta t \sigma_x^n(i, j)}{2\mu(i, j) + \Delta t \sigma_x^n(i, j)} \\
C_{hxy}(i, j) &= \frac{2\Delta t}{\left[ 2\mu(i, j) + \Delta t \sigma_x^n(i, j) \right] \Delta x} \\
H_y^{n+\frac{1}{2}}(i, j) = C_{hh}(i, j) H_y^{n-\frac{1}{2}}(i, j) + C_{hxy}(i, j) \left[ E_x^n(i, j + 1) - E_x^n(i, j) \right] \\
C_{hh}(i, j) &= \frac{2\mu(i, j) - \Delta t \sigma_x^n(i, j)}{2\mu(i, j) + \Delta t \sigma_x^n(i, j)} \\
C_{hxy}(i, j) &= \frac{2\Delta t}{\left[ 2\mu(i, j) + \Delta t \sigma_x^n(i, j) \right] \Delta y} \\
H_z^{n+\frac{1}{2}}(i, j) = H_z^{n+\frac{1}{2}}(i, j) + H_y^{n-\frac{1}{2}}(i, j)
\]

Figure 3.4: (a). Solutions to Maxwell’s equations in media with anisotropic conductivities (b). Simplified solutions under the matching condition (c). The split-PML scheme (d). Modified update equations
Figure 3.5: (a). Scattering from circular PEC object (b). Scattering from rectangular PEC object (c). Scattering from a wedge Dielectric object
Chapter 3. Emulation of the FDTD: RNNs

Figure 3.6: (a). TF/SF boundary (b). Consistency updates

Consistency Updates

O-1 Region
\[ H_{x}^{n+1/2}(i,j) = H_{x}^{n+1/2}(i,j) - C_{x,y}(i,j) E_{y,n}^{n}(i,j) \]

O-2 Region
\[ E_{x}^{n+1}(i,j) = E_{x}^{n+1}(i,j) + C_{x,y}(i,j) H_{y,n}^{n+1/2}(i,j) \]
\[ H_{x}^{n+1/2}(i,j) = H_{x}^{n+1/2}(i,j) + C_{x,y}(i,j) F_{y,n}^{n}(i,j) \]

O-3 Region
\[ E_{x}^{n+1}(i,j) = E_{x}^{n+1}(i,j) + C_{x,y}(i,j) H_{y,n}^{n+1/2}(i,j) \]
\[ H_{x}^{n+1/2}(i,j) = H_{x}^{n+1/2}(i,j) + C_{x,y}(i,j) F_{y,n}^{n}(i,j) \]

O-4 Region
\[ H_{x}^{n+1/2}(i,j) = H_{x}^{n+1/2}(i,j) - C_{x,y}(i,j) E_{y,n}^{n}(i+1,j) \]

I-1 Region
\[ E_{x}^{n+1}(i,j) = E_{x}^{n+1}(i,j) - C_{x,y}(i,j) H_{y,n}^{n+1/2}(i,j) \]

I-4 Region
\[ E_{x}^{n+1}(i,j) = E_{x}^{n+1}(i,j) - C_{x,y}(i,j) H_{y,n}^{n+1/2}(i-1,j) \]

Figure 3.6: (a). TF/SF boundary (b). Consistency updates
Figure 3.7: (a). Scattering from circular PEC object (b). Scattering from rectangular PEC object (c). Scattering from a wedge Dielectric object
Chapter 3. Emulation of the FDTD: RNNs

3.2 The Dataset

Three separate datasets for experiments are created using an in-house developed FDTD solver for the TE\(^z\) mode. Each dataset comprises 100 simulations with a Gaussian pulse which has the highest frequency component of 2 GHz. In each simulation all three field components \(E_x\), \(E_y\) and \(H_z\) are recorded within a region of 128 \(\times\) 128 cells for 400 time steps and finally, all the frames in which the excitation hasn’t yet fully entered the computational domain are cut from its beginning. This ensures that examples that are presented to the network contain meaningful information.

For dataset Type-1, the TF/SF excitation is used to sweep a planar wavefront (angle of propagation chosen randomly between 20° and 70°) across the domain of interest and scattered waves from PEC objects are observed. For dataset Type-2, a point source is applied at a random location resulting in a spherical wavefront which then propagates across the domain of interest and scatters from PEC objects. For these two datasets, PEC objects are a random mix of circular and square shapes with sizes chosen randomly between 0.4\(\lambda_{\text{min}}\) to 0.6\(\lambda_{\text{min}}\). Finally, for dataset Type-3, two point sources are applied at random locations resulting in a complex wavefront pattern (formed by the sum of the fields from each source at each point in time) which then propagates across the domain of interest and scatters from a circular PEC object of fixed size and fixed location (at the bottom left corner of the domain).

In each dataset type, 75 simulations are allocated to training and 25 to testing. The size of the training sets depends on the size of the videos (which is a hyperparameter). The total number of simulated frames available for training for dataset Type-1 is 18,750 and for datasets Type-2 and Type-3 is 15,750.
Chapter 3. Emulation of the FDTD: RNNs

3.3 The Architecture

The model’s architecture is shown in Fig. 3.8. It comprises a convolutional encoder, a convolutional LSTM [38] and a convolutional decoder [39] and consumes data in the form of videos (subsets of simulations). When a video is presented to the network, the convolutional encoder extracts its first frame and, through the multi-layer convolutional operations, compresses the spatial domain. The features extracted from the first frame of the video by the encoder are fed to the RNN. Subsequently, the hidden state of the RNN is recursively updated for a specific number of times to produce a stack of representations of the temporal field evolution. Finally, the stack of updates is fed to the decoder to construct complete future frames of the EM fields for that video. To incorporate the object information into the learned representations, and to ensure that the objects remain in their positions while the dynamics of wave propagation and scattering evolve, we employ the background-foreground separation strategy [40]. The object information is provided as a binary background (1’s for locations where the PEC object exists and 0’s elsewhere) together with input time-domain field data.

The convolutional encoder and decoder are both implemented using residual blocks. Deep residual learning framework (ResNet) was recently proposed to alleviate the problem of accuracy degradation in training very deep networks [41]. In this work, residual blocks are constructed from convolutional layers based on Visual Geometry Group (VGG) design guidelines for large-scale visual recognition [42]. Additionally, for the network to be able to generate predictions that resemble the training examples closely, it is essential that the decoder ends with a linear convolutional layer. The convolutional LSTM proposed in [38] is modified to incorporate the background-foreground strategy discussed above. The feature maps extracted from the geometry are split in two segments $B_{\text{mul}}$ and $B_{\text{add}}$ that are then mixed with the hidden state (which is the latent representation of the fields) before each update. This process is
Chapter 3. Emulation of the FDTD: RNNs

Figure 3.8: (a) and (b). General overview (c). The modified LSTM cell shown in Fig. 3.8(c). For the first update, $c_{t-1}$ and $h_{t-1}$ are initialized to zero and $x_t$ is the features extracted from the fields by the encoder; for all succeeding updates, $x_t$ is set to zero and $c_t$ and $h_t$ are updated from $c_{t-1}$ and $h_{t-1}$. The stack of hidden states $h$ is interpreted as the latent representation of the future time-steps.
Chapter 3. Emulation of the FDTD: RNNs

During training, the generated (predicted) time-domain results is compared with the original input time-domain data using a tailored loss function to drive the optimization process. The loss function includes two terms: the Frobenius norm and the Gradient Difference Loss (GDL) [43]. The latter term contributes to sharpness of the predicted frames which in our application is useful because it enforces clear boundaries between objects and free space regions. The GDL is added to the norm term with an experimentally determined coefficient.

3.4 Numerical Experiments

The FDTD simulations for the dataset generation are run on a Desktop PC with Intel Core i7-6700 Processor (3.4 GHz) and 4 GB RAM, with the ‘Number of Cells per Wavelength’ parameter set to 40 (the entire computational domain is a square with length of 1.5 m; Courant stability factor is set to 0.7071).

For efficient training it is essential that the datasets be properly scaled and, with the raw simulations, there are two main problems in the way of achieving that. The first problem is that the magnitudes of the electric and magnetic fields are not on the same scale; in fact they differ by a factor of $\eta$ (wave impedance). As a solution to this problem we scale up the magnetic field with the wave impedance with the implication that our network learns $\eta H_z$ (and not $H_z$ directly). The second problem is that the usual normalization or standardization methods become problematic when dealing with three-channeled data which represent physical vector fields. This is due to the fact that the addition of a constant and the scaling of each channel (which represents a component of a vector field) independently will distort the spatial distribution of that vector field and therefore distort the ‘ground truth’ that is being presented to the network. To overcome this difficulty, we adopt a special scaling scheme in which all channels ($E_z$, $E_y$ and $\eta H_z$) are scaled with a single constant value. To obtain this
constant, first, the maximum of the magnitudes of the electric and scaled-up magnetic fields in the entire dataset are found and then the maximum (or average) of these
two values are chosen as the single constant scaling factor. With this scheme the magnitudes of the fields will be distributed around unity and the spatial distribution of the fields will remain undisturbed. It is noted that the scaling of datasets is one of the pre-processing steps and it is applied prior to the splitting into training and testing portions.

Training was done using an NVIDIA Tesla K40M GPU. For dataset Type-1, 150 frames are cut from its beginning (bringing the total number of frames to 250) and video size is set to 25 frames. For datasets Type-2 and Type-3, 190 frames are cut from their beginning (bringing the total number of frames to 210) and video size is set to 30 frames. The EM field data in the spatial domain is first compressed via the convolutional encoder, and then fed into the RNN. Network is trained for 1000 Epochs which took around 48 hours. In the inference phase, we use the trained network as a predictive model to emulate transient EM scattering problems with randomly distributed PEC objects on the same Desktop PC that was used to generate the datasets. We adopt two strategies for generating a complete predicted simulation from testing scenarios. The first one, which we call Test Prediction, corresponds to the training pipeline; in other words, network predicts videos of a simulation separately and the predicted videos are put together to form a complete predicted simulation. In the second strategy, which we call Continuous Prediction, network predicts the first video of a simulation and then the last frame of this initial prediction is extracted to seed the prediction of the next video, forming a closed loop from network’s output to its input.

The physical phenomena involved in scattering from circular shapes are reflection and creeping waves; for rectangular shapes with edges and corners, they are reflection and diffraction. As can be seen from these figures, in each scenario the predicted frames are almost identical to the ground truth; defining the error, for each frame, as the difference between the ground truth and prediction for that frame, we observe that its magnitude falls within 5% of the maximum of its Frobenius norm over the entire
Chapter 3. Emulation of the FDTD: RNNs

simulation (for Continuous Prediction mode, this is true as long as the wave has not completely exited the computational domain; in other words, as long as the fields are not completely zero). We argue that our network has learned the fundamental transient behaviors of wave propagation and scattering.

3.5 ERD Model and DDM Parallelization

Implementation of the DDM-based parallelization involves three conceptual steps. In the first step, following the Method of Lines (MoL) [44, 45, 46], the Maxwell’s equations, over the domain of interest, are cast in the form of a time-dependent system of linear equations. The solution to this system can be expressed as the multiplication of the matrix exponential with the vector of initial conditions. However, due to their prohibitive computational cost, the matrix exponential operations will require some form of preconditioning.\footnote{The solution to a linear system of equations, described by $Ax = b$, is given by $A^{-1}b$. However, explicit computation of $A^{-1}b$ is, in general, expensive and inefficient. One approach to efficient computation of $A^{-1}b$ is \textit{preconditioning} which involves replacing $A$ with a \textit{preconditioning matrix} $M$ (chosen such that it is, in some sense, similar to $A$) for which computation of the \textit{preconditioning operation} $M^{-1}b$ is much more efficient.} In the second step, a block-diagonal matrix is chosen as the preconditioner for the system of equations. With this choice, the non-zero elements of the matrix exponential are in the form of diagonal-blocks that are themselves matrix exponentials albeit much smaller in size. Therefore, using a block-diagonal preconditioner will transform a matrix-vector multiplication involving a large matrix exponential into a set of parallel matrix-vector multiplications involving smaller matrix exponentials. In the final step, a generalized Runge Kutta (RK) method, which is based on preconditioning operations, is used for integration [47]. The block-wise preconditioning operations are parallel evaluations of sub-domain solutions and can, therefore, be replaced with the learned ERD model.
Chapter 3. Emulation of the FDTD: RNNs

Figure 3.10: Results of numerical experiments with datasets Type-4 and Type-5.

Figure 3.10: Results of numerical experiments with datasets Type-4 and Type-5.
Chapter 3. Emulation of the FDTD: RNNs

To assess the feasibility of this scheme, we setup an experiment in which the ERD architecture is first trained with sub-domain FDTD videos from parallelized simulations of propagation and scattering events within a domain of certain size. Then, in inference phase, its set of next-frame sub-domain predictions are used in the generalized RK algorithm to compute the next temporal state for similar events in domains of larger size. To this end, the subdomain data from $256 \times 256$ simulations are used to train the ERD model. In the inference phase, the model’s effectiveness is evaluated by (1) applying it to test scenarios of the same dimensions and (2) applying it to $512 \times 512$ simulations (which were never seen during training).

The in-house developed FDTD solver is used to generate the datasets required for this experiment. Dataset Type-4 contains two-time-step simulations of size $256 \times 256$ (which can be divided into 4 subdomains of size $128 \times 128$); In this dataset, a single point source excites a spherical wavefront which scatters from a single circular PEC object of size $0.5\lambda_{min}$ and eventually dissipates in the PML walls. The relative locations of the point source and the PEC object were changed based on all unique combinations for which source and object remained completely in a subdomain; In the interval between frames 180 to 250, every five frames, two frames were recorded resulting in a total of 90 simulations. 67 simulations were randomly selected for training and the rest were assigned to testing. Dataset Type-5 contains two-time-step simulations of size $512 \times 512$ (which can be divided into 16 subdomains of size $128 \times 128$); In this dataset, a single point source, the location of which is fixed at the lower right of the domain, excites a spherical wavefront which scatters from a single circular PEC object of size $0.5\lambda_{min}$ (the location of which is fixed within a subdomain near the middle of the domain) and eventually dissipates in the PML walls. In the interval between frames 320 to 450, every ten frames, two frames were recorded resulting in a total of 14 simulations for testing.

The results of the experiments are shown in Fig. 3.10. The run time of the DDM parallelization scheme exhibits 2x speedup when the learned model is used for sub-
Chapter 3. Emulation of the FDTD: RNNs

domain computations compared to its pure form which uses matrix exponential operations. This experiment successfully demonstrates that the learned model of EM dynamics can be incorporated as a valuable building block into the conventional computational workflows.
Chapter 4

Emulation of the FDTD: Transformers and GNNs

In Thrust-I of my research, the goal is to leverage the readily available simulation data to characterize the physical processes involved in propagation, reflection and scattering of electromagnetic waves using suitable state-of-the-art models. As discussed in the previous chapter, we successfully demonstrated that a convolutional Encoder-Reccurrent-Decoder (ERD) architecture can be used as a data-driven and learned model for emulation of electromagnetic dynamics. Furthermore, by successfully applying this learned model to a Domain Decomposition (DDM) parallelization scheme, we also demonstrated that this learned model of EM dynamics can be incorporated as a valuable building block into the conventional computational workflows. In the above parallelization scheme, the task of the learned model is to provide the next time-step in each sub-domain. While the ERD architecture proved to be a decent candidate, it might not be the only one suited for this task. Recently, new architectures have been demonstrated for processing of sequential data. One example is the Transformer [48] which was developed in the field of Natural Language Processing for translation tasks. In that context, it takes a segment of text in one
Chapter 4. Emulation of the FDTD: Transformers and GNNs

language and outputs its translation in another. To represent text numerically, it is first divided into subsegments called Tokens (which can be words, sub-words or even characters); These are then represented with high-dimensional vectors using a process called Embedding. As such, both the input and the output (target) of the Transformer are sequences of vectors. Another example is the Graph Neural Network (GNN) [49, 50] which was developed to handle graph-structured data such as connectivity in social networks, arrangement of atoms in a molecule, etc. A graph comprises a set of nodes and a set of edges. Each node has a vector of attributes associated with it. An edge represents a relation between the two nodes that it connects and has a weight factor associated with it. The choice of node attributes and edge weight factors depend on the application. GNN constructs representations from node attributes, while taking into account the edge weight factors. These representations can then be used to generate some outputs. Regardless of its original domain of application, GNN can also be applied to unstructured data types such as images and texts by interpreting them as graphs. In the case of an image, the pixels can be represented by the nodes and the spatial adjacency of the pixels can be represented by edges while the red, green, and blue pixel values constitute the vector of attributes for each node. In the case of a text, the tokens can be represented by the nodes and the adjacency of tokens in a sentence (or any syntactic/semantic relation between them) can be represented by edges while the embeddings constitute the vector of attributes for each node. A Convolutional Graph Neural Network (CGNN) constructs representations by updating the attribute vector of each node using information from its neighbors. First, each node prepares a message for each of its neighbors from its current state of attribute vector and then it passes those messages to their intended recipients. Successive operation of a CGNN on a graph produced a sequence of attribute vectors. As such, the output of a CGNN is a sequence of vectors.
4.1 **The Architecture: Transformer**

In this work we use the Transformer with minimal modifications for the task of time-series forecasting. Therefore, initially, we describe the architecture in the context of language translation and then introduce modifications that are required to adapt it to our specific application, i.e., predicting future frames in FDTD videos. To clarify concepts that are, unavoidably, carried over from the field of NLP, we must mention that text is represented numerically by first dividing it into sub-segments called tokens (which can be words, sub-words or even characters) [51] and then embedding each token, i.e., representing each token with a high-dimensional vector [52]. Therefore, through tokenization and embedding, a segment of text can be represented as a sequence of vectors.

Operation of the transformer in inference phase is shown in Fig. 4.1(a). The segment of text in original language is transformed into a sequence of embeddings (henceforth referred to as the context sequence) and positional encoding is added to give the model information regarding the relative or absolute position of the elements in the sequence. The result is then fed to the encoder which will map it to a representation sequence. Output generation is an auto-regressive process. It starts with feeding the decoder with a one-element input sequence containing the Start-of-Sequence (SoS) symbol along with the encoder’s output. The last element in the decoder’s output is extracted and then passed through softmax and argmax to get the most probable token, i.e., to get the token that is most probable to come next in the output sequence. This token is then again embedded and appended to the input sequence. This process continues with the decoder taking in the entire previously predicted output sequence along with encoder’s output in each step until the End-of-Sequence (EoS) symbol is predicted. It must be noted that, in each step, decoder’s output sequence has the same number of elements as its input; In the training phase decoder has learned to shift input sequence elements forward by one position.
Chapter 4. Emulation of the FDTD: Transformers and GNNs

Figure 4.1: (a) Transformer in the inference phase. (b) encoder and the decoder operations. (c) self-attention operations; Dotted lines with no label represent reshape operation.
Therefore, in each step of output generation, only the last element in the output sequence contains new information. The internal operations of the encoder and the decoder are shown in Fig. 4.1(b). They both consist of stacked layers. Each encoder layer consists of a block of self-attention operations and a fully connected Feed-Forward Network (FFN). Each decoder layer consists of two types of self-attention operations and a fully connected FFN. Self-Attention operations are shown in Fig. 4.1(c). Self-Attention produces a representation of the relatedness of different positions of a sequence without regard to their distance. These operations can be described as mapping a set of Query, Key and Value vectors to a set of output vectors. For each position in the sequence, first, the dot products of that position’s Query with the Keys of all other positions are computed. Then, a softmax function is applied to the resulting vector. Finally, the dot product of the results with the Value vector of that position produces the attention output for that position. In practice, The Queries, Keys, and Values for all positions are packed into matrices which enables simultaneous computation of attention for all positions of the sequence. To allow the model to jointly attend to information from different representation subspaces at different positions, the Queries, Keys and Values are broken down, and Self-Attention is computed in parallel for each sub-segment (i.e., Attention Heads) and the outputs are then concatenated. It must be noted that, based on how the Queries, Keys, and Values are constructed and whether a Look-Ahead-Mask is used, three variations of Self-Attention can be found in encoder and decoder layers. The Look-Ahead-Mask, which is used in the causal-self-attention block of each decoder layer, enforces causality in the decoder, i.e., it prevents each position of the input sequence from attending to its subsequent positions.

Since our data are sequences of vectors (i.e., sequences of 2D field distributions that are reshaped into sequences of 1D vectors) they are already in numeric form, therefore, we remove the tokenization and embedding from the architecture. Since the model’s predictions for our data will be continuous values representing physical quan-
Chapter 4. Emulation of the FDTD: Transformers and GNNs

tities (as opposed to its output for text data which is a probability distribution) we also remove the softmax layer from the architecture. Furthermore, we divide each sequence into past and future segments. The past segment plays the role of the context sequence and is fed to the encoder. Also, its last frame is fed to the decoder as the initial frame for the prediction of the future sequence. With this proposed scheme, we translate the past into the future using the Transformer architecture.

4.2 The Architecture: GNN

An example of a graph is shown in Fig. 4.2(a). It comprises a set of nodes and a set of edges. Each node has a vector of attributes associated with it. An edge represents a relation between the two nodes that it connects and has a weight factor associated with it. The choice and physical meaning of node attributes and edge weight factors depend on the application. GNN constructs representations from node attributes, while taking into account the edge weight factors. These representations can then be used to generate some outputs. Regardless of its original domain of application, GNN can also be applied to unstructured data types such as images and texts by interpreting them as graphs. In the case of an image, the pixels can be represented by the nodes and the spatial adjacency of the pixels can be represented by edges while the red, green, and blue pixel values constitute the vector of attributes for each node.

The Graph Convolution Layer (GCL) operations are shown in Fig. 4.2(b). First, each node prepares a message for each of its neighbors from its current state of attribute vector and then it passes those messages to their intended recipients. Once messages from all their neighbors are received, each node will aggregate them (by applying a function such as sum, average, etc.) and then use the resulting representation to update the current state of its attribute vector (by concatenation, RNN, etc). Successive operation of a GCL on a graph produces a sequence of attribute vectors.
Chapter 4. Emulation of the FDTD: Transformers and GNNs

Figure 4.2: (a) definition of a graph. (b) Convolutional Graph Layer updates the state of each node’s attribute vector. (c) emulation of the FDTD using message passing.

The 2D TE$^2$ Yee grid and its update equations are shown in Fig. 4.2(c). This operation bears a striking resemblance to the operations of a GCL. As shown in Fig. 4.2(c), we propose to represent the Yee grid as a graph in which nodes are the cells, node attribute vectors comprise the three field components $E_x$, $E_y$ and $H_z$ and edges connect neighboring cells. As nodes exchange information (in the form of messages) with each application of the GCL to the graph, fields will progress forward in time by one step.
Chapter 4. Emulation of the FDTD: Transformers and GNNs

4.3 Numerical Experiments

The FDTD simulations for dataset generation and model inferences were run on a Desktop PC (Intel Core i7-4790, 3.6 GHz; 32 GB of RAM; Windows OS) while training sessions were run on an NVIDIA Tesla K40M GPU. We use the same data scaling scheme and loss function as in the previous work [37]. The dataset was preprocessed by scaling all frames with the maximum of the field magnitudes. The loss function comprises two components the first of which is the Frobenius norm of the difference between the true and predicted frames. To further clarify, let’s denote a frame from the FDTD solver at time $t$ by $f^\text{FDTD}_t$ and denote its corresponding frame generated by the model by $f^\text{MODEL}_t$. Then training error for each frame is constructed as $\|f^\text{FDTD}_t - f^\text{MODEL}_t\|_{\text{Frob}}$. The second component, is the Gradient Difference Loss (GDL) [43]. The latter term contributes to sharpness of the predicted frames which in our application is useful because it enforces clear boundaries between objects and free space regions. For dataset Type-1, the video size is set to 10 frames resulting in training set size of 1,875 examples. For datasets Type-2 and Type-3, it is set to 15 frames resulting in training set size of 1,050 examples. To assess the accuracy of each model’s predictions, in the inference phase, we measure $|f^\text{FDTD}_t - f^\text{MODEL}_t|^2$ for each time step and plot it alongside the true and predicted frames for that time step. This results in a sequence of accuracy maps which illustrates the quality of predictions.

Since the purpose of the experiments reported in the present work is to compare and contrast the performance of Transformers and GNNs with that of the ERD, we use the color scale of the accuracy maps to link the performance of these models. Specifically, we clamp the color scale of the accuracy maps for both the Transformer and the GNN to the same value as that for the ERD accuracy maps. In this way, it can be argued that both plots are being viewed on the same color scale and can, therefore, be compared. Furthermore, in each simulation, the value that is used to
Figure 4.3: Transformer’s predictions in the Continuous Prediction mode
Chapter 4. Emulation of the FDTD: Transformers and GNNs

Figure 4.4: CGNN’s predictions in the Test Prediction mode

Figure 4.4: CGNN’s predictions in the Test Prediction mode
clamp the color scale, denoted by $N_{\text{Frob,max}}^{\text{ERD}}$, is the maximum of the Frobenius norm of the error for predictions of the ERD model, i.e., $N_{\text{Frob,max}}^{\text{ERD}} = \max(\|f_t^{\text{FDTD}} - f_t^{\text{ERD}}\|)$. To assess the speedups enabled by our proposed models, we also compare our model’s average per-frame inference latency with the average time for a single update of an open-source FDTD solver. We use MEEP [53] to simulate wave propagation and scattering in free space and in presence of a circular PEC object. All relevant simulation parameters such as domain size, number of cells and pulse frequency are kept the same as for simulations with our in-house developed solver.

In training the Transformer, we adopt the teacher forcing strategy [54, 55], i.e., the model is fed with an input sequence consisting of the true frames instead of its own previous predictions. This approach contributes to robustness and speed of the training process. The hyperparameter values in our implementation are as follows: both the encoder and the decoder consist of 3 layers; $d_m = 256$ and $n_h = 8$. This configuration results in the largest possible model that we can implement using a single GPU. Larger values for the above mentioned hyperparameters will cause the GPU to throw an OOM (Out Of Memory) error and abort the training. The size of the implemented model with the configuration mentioned above is about 60M (million) parameters. For training with dataset Type-1, each video is split such that the first 5 frames represent the past (context) sequence (i.e., $l_c = 5$) and the remaining 5 frames represent the future (input) sequence. Likewise, for training with datasets Type-2 and Type-3, each video is split such that the first 10 frames represent the context sequence ($l_c = 10$) and the remaining 5 frames represent the input sequence. Models are trained for 500 epochs which takes about 10 hours. In inference phase, the last frame of the past sequence is used as the initial frame of the future sequence and the decoder predicts the rest of it autoregressively. For the accuracy of the predictions, we observe that $|f_t^{\text{FDTD}} - f_t^{\text{TRNSF}}|^2 \leq 0.05 \times N_{\text{Frob,max}}^{\text{ERD}}$. When measuring the inference latency (i.e., the time it takes for the model to make predictions once it is fed with the inputs), we observe that simultaneous prediction of multiple videos...
(i.e., batching multiple videos) results in lower average per-frame inference latency. Specifically, the average per-frame inference latency for prediction on batches of 5, 10 and 25 videos is 123.37 ms, 63.33 ms and 25.30 ms respectively. Furthermore, these values are unaffected by the length of the context sequence (or equivalently, the length of the predicted sequence). Therefore, prediction using the Transformer can be very fast if a large number of videos are batched together for processing (as many as can fit into the memory). Since the average time for a single update for our in-house developed FDTD solver and MEEP are 373 ms and 42.61 ms respectively, the Transformer exhibits speedups of $14\times$ and $1.5\times$, respectively, with a batch size of 25.

Samples from predicted frames are shown in Fig. 4.3. While the model has learned to propagate the wavefronts correctly, it has trouble recognizing the shape, size and location of objects. For example, in the Type-1 right column sample, the rectangular shape of the lower object is misrepresented as a circle, or in the Type-2 middle column sample, extra objects (i.e., non-existent in the true frames) of both circular and rectangular shapes are predicted. We had encountered a similar problem when developing the ERD model; In that work, we were able to satisfactorily eliminate it by enforcing a foreground background scheme, i.e., by implementing a second branch for the encoder which processed only the object information and produced representations that were mixed with the output of the LSTM at each time-step. We implemented a similar solution for the Transformer but it did not alleviate the problem, which hints at the fact that the issue could be a manifestation of a more fundamental problem. A glance at the literature, reveals that Transformer models that exhibit state-of-the-art performance are much larger in size and are trained with much larger training sets compared to our implementation. For example, GPT [56] consists of 12 decoder layers, $d_m = 768$ and $n_h = 12$ resulting in a total size of 117M (million) parameters; GPT-2 [57] consists of 48 decoder layers, $d_m = 1600$ and $n_h = 12$ resulting in a total size of 1.5B (billion) parameters. In terms of data, for
Chapter 4. Emulation of the FDTD: Transformers and GNNs

GPT, $l_c = 512$ and for GPT-2, $l_c = 1024$. Therefore, we are inclined to conclude that small model size and small training example size are the main reasons behind the shortcomings in prediction performance of our implementation.

For the GNN, since our frame size is $128 \times 128$ pixels, $n_N = 16,384$ and $n_E = 65,024$. Therefore, we set $d_m = 32$ which is the largest possible model size that we can implement using a single GPU. Larger values for $d_m$ will cause the GPU to throw an OOM (Out Of Memory) error and abort the training. The size of the implemented model with the configuration mentioned above is about 13K parameters. The update operation of our GCL implementation is an RNN which produces a new attribute vector from the previous attribute vector and the representation obtained from aggregated messages. The first phase of our experiments with the GCL was focused on training it in the Continuous Prediction mode, i.e., in each training step we fed it with a single true frame and fed back its output to its input multiple times and compared the predictions with true frames in the loss function. The GCL was unable to learn useful representations of EM dynamics and field distributions of its predictions decayed to zero after a few frames. We believe that its very small size is the main reason the model is incapable of learning in the Continuous Prediction mode. Therefore, in the second phase, we changed our focus to the Test Prediction mode and required that it only predict the next frame. Models are trained for 250 epochs which take about 30 hours. For the accuracy of predictions, we observe that $|f^{FDTD}_t - f^{GNN}_t|^2 \leq 0.05 \times N_{Frob,max}^{ERD}$. The average per-frame inference latency is 43.17 ms. Since the average time for a single update for our in-house developed FDTD solver and MEEP are 373 ms and 42.61 ms respectively, the GNN exhibits a speedup of $9 \times$ compared to the former and no speedup compared to the latter. Samples from predicted frames are shown in Fig. 4.4. Apart from a scaling factor, the predictions are in good agreement with the true frames.
Chapter 5

Device Modeling: GANs

Given some objectives, the first step in designing an antenna is to acquire estimates for its structural and material parameters using formulas that link these parameters to antenna’s radiation characteristics. These analytical expressions are derived from application of Maxwell’s equations to the structure under investigation. While the radiation characteristics of an antenna that is built with these estimates will be close to the desired values, some fine-tuning is required before they precisely match the prescribed objectives. Therefore, the second step in the design process is fine-tuning the initial estimates. Traditionally, this was achieved through multiple cycles of fabrication, measurement, and corrections. However, with the aid of modern powerful computers and sophisticated simulation software, fine-tuning is now achieved using computer simulations. Using a Computer Aided Design (CAD) software, the space of parameters is gridded (with sufficient resolution) in a neighborhood of the initial estimates and a full-wave simulation is carried out for each point of that grid to obtain its corresponding radiation characteristics. The results are inspected and the point (in the parameter space) which corresponds to the best match is chosen as the final design. Therefore, the design process involves both a theoretical model and an experienced designer who is able to not only bring the theory to bear upon
the particular application but also to guide the fine-tuning process until the optimal candidate is found. The design process can also be automated using an iterative procedure known as the inverse design. In this setting, the initial estimates can be chosen randomly, then the corresponding radiation characteristics are computed and are compared to the objectives to calculate a change to the parameters. Parameters are updated and computations are repeated until the prescribed objectives are met. In any case, the product of the design process contains both theoretical and practical knowledge regarding optimal designs for the particular application and committing it to a database ensures that the invested knowledge is not lost and will be accessible in the future. Furthermore, in applications with similar objectives, this knowledge can be drawn upon to increase the efficiency of the design process by pulling that design from the database and using it as a starting point for development.

The output of the design process, whether it is driven by a designer or an algorithm, is a collection of near-optimal designs. From a probabilistic point of view, there is a data-generating distribution that these designs (data-points) are drawn from. If this distribution is known, the designs (data-points) that are drawn from it will conform to the objectives of the particular application. Therefore, knowing the data-generating distribution is equivalent to having a database which contains the combined knowledge and experience that was invested in creating those near-optimal designs in the first place.

With the aim of constructing a data-driven and learned knowledge database, we investigate whether a Generative Adversarial Network (GAN) [58] can learn the distribution of a dataset of near-optimal antenna designs. In other words, whether a GAN can be asked to produce new antenna designs that exhibit a desired response [59, 60]. As an example, we consider the problem of designing the Log-Periodic Folded Dipole Array (LPFDA) antenna [61] for two non-overlapping ranges of Q-factor values. By representing the antenna with the vector of its structural parameters and considering each desirable range of the Q-factor as a class, we transform our problem to that
Chapter 5. Device Modeling: GANs

of generating new samples from a given class. We set up experiments in which a Conditional GAN (CGAN) [62] and a library of vanilla GANs are first trained with a dataset of near-optimal structures and then, in inference phase, are asked to generate new samples (LPFDA designs) from each class (Q-factor range).

The dataset for experiments comprises vectors of structural parameters for LPFDA designs that exhibit desirable Q-factor values in certain UAV applications. A CAD software was used to generate a multitude of designs for two non-overlapping Q-factor ranges (labeled as class and ) by randomly changing certain geometric features within some established bounds.

We experiment with two different architectures that we implemented in TensorFlow. The first is a library of vanilla Wasserstein GANs (WGANs) [63]. In this case, a separate vanilla WGAN is trained for each class and in inference phase the appropriate one is chosen to generate examples from the desired class. The second is Conditional WGAN. In this case, in contrast to the vanilla WGAN, the Generator can be instructed to generate examples from a specific class by passing to it the label of that class along with the noise vector. For both vanilla GAN and CGAN we have used the Wasserstein loss variants as they are significantly more stable during training.

In inference phase, to measure the accuracy of the GAN’s predictions, we ask both models to produce some new designs for each class and then compute the associated Q-factor value using the CAD software. It is observed that the accuracies of the generated designs for classes 0 and 1 are 62% and 80% in the case of the library of vanilla WGANs and are 80% and 98% in the case of the Conditional WGAN. These experiments successfully demonstrate that these architectures, especially the Conditional WGAN, can be utilized as a data-driven and learned knowledge database and can be relied upon for fast generation of suitable designs.
Chapter 5. Device Modeling: GANs

5.1 The Dataset

The geometry of the LPFDA is shown in Fig. 5.1(a). The array is designed to operate from 350 MHz to 1 GHz and its elements are printed on a substrate with $\epsilon_r = 2.33$ and $h = 0.786$ mm. The structure is fed from the bottom using a coaxial cable that drives the center conductor. Ten structural parameters were used to form the antenna’s vector representation $X = [x_2, x_3, x_4, x_5, c_2, c_3, c_4, c_5, a_1, a_2]^T$ while $x_1$, $x_6$, $x_7$, $c_1$, $c_6$, $c_7$ and $c_8$ were not used in the dataset and were kept constant in the design ($x_1$ and $c_1$ fix the lower half of the bandwidth).

The Q-factor of an antenna can be calculated from its input impedance, i.e., $Z(\omega_0) \equiv R(\omega_0) + jX(\omega_0)$, using $Q(\omega_0) = \frac{\omega_0}{2R(\omega_0)} \sqrt{(R'(\omega_0))^2 + (X'(\omega_0) + \frac{|X(\omega_0)|}{\omega_0})^2}$ in which the prime denotes differentiation with respect to frequency [64]. Since LPFDA is a wide-band antenna, we define its Q-factor as the average of Q-factors at several frequencies sampled in its operational bandwidth.

Three datasets were constructed based on the ranges of interest for Q-factor values. In the first dataset, named $rfant\ 1$, $40 < Q < 80$ and in the second dataset, named $rfant\ 2$, $Q < 40$. These two datasets were used to train each vanilla GAN separately. The third dataset, named $rfant\ 3$, is the union of the previous two, with label 1 assigned to examples from $rfant\ 1$ and label 0 assigned to examples from $rfant\ 2$. This dataset was used to train the CGAN (it must be noted that these 0 and 1 labels denote the class of data and are different from those that are used in the loss function to designate fake and real examples). To generate the datasets, CST Microwave Studio was used to simulate 500 designs (with an average run time of half an hour per design) for each of $rfant\ 1$ and $rfant\ 2$ by randomly selecting values for each parameter from a certain range as shown in Fig. 5.1(b).
Figure 5.1: (a) the geometric structure of the LPFDA (b) a histogram showing the distribution of values in both \textit{rfant 1} and \textit{rfant 2}.

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|}
\hline
\textbf{Parameter} & \textbf{Range (mm)} \\
\hline
x1 & 34.99 & -- \\
x2 & 29.00 & 38.00 \\
x3 & 26.00 & 37.00 \\
x4 & 23.00 & 30.00 \\
x5 & 22.00 & 28.00 \\
x6 & 19.30 & -- \\
x7 & 18.30 & -- \\
c1 & 15.83 & -- \\
c2 & 13.00 & 18.00 \\
c3 & 12.00 & 15.00 \\
c4 & 9.00 & 12.00 \\
c5 & 8.00 & 12.00 \\
c6 & 7.20 & -- \\
c7 & 6.00 & -- \\
c8 & 4.60 & -- \\
a1 & 1.50 & 2.70 \\
a2 & 5.50 & 6.50 \\
\hline
\end{tabular}
\end{table}
Chapter 5. Device Modeling: GANs

5.2 The Architecture

GANs can learn an estimate of the probability distribution of a given dataset. They comprise two models, a Generator and a Discriminator, that are trained simultaneously in an adversarial game. GAN’s architecture in the training phase is shown in Fig. 5.2(b). The Generator maps a vector of random noise to a vector of features. The Discriminator decides if its input is from the dataset (real) or is produced by the Generator (fake). Therefore, in the training phase, the Generator learns to deceive the Discriminator by producing features that increasingly resemble the ones from the dataset. In other words, in the training phase, the Generator learns an estimate of the distribution of the dataset and can map random noise to examples from that dataset. GAN’s architecture in the inference phase is shown in Fig. 5.2(a). In the inference phase, the Generator will produce new examples that are similar to the ones from the dataset as they are drawn from the same distribution. The Vanilla GAN has no mechanism for guiding the generation of datapoints. In other words, in the case of a dataset that comprises multiple categories of examples, the Vanilla GAN can’t be instructed to generate datapoints from a specific category. The Conditional GAN was introduced to address this problem. The generator of the CGAN accepts a noise vector (which is sampled from the normal distribution) and a binary class label as inputs. The label is provided as an integer and is then embedded (i.e., projected onto a higher-dimensional space) and concatenated with the noise vector. This mixture of noise and class information is mapped to a datapoint after passing through the Generator.

The Wasserstein GAN (WGAN) provides a significant improvement in training stability by (1) replacing the original GAN’s loss with Wasserstein loss as a measure of the closeness of the predicted distribution and data distribution (2) constraining the Discriminator to the set of 1-Lipschitz Continuous functions by confining (clipping) its weights to a compact space $[-c, c]$ (3) updating the Discriminator weights...
Chapter 5. Device Modeling: GANs

Figure 5.2: (a). GAN and CGAN in inference phase (b). GAN and CGAN in training phase

to optimality before each update of the generator weights and (4) using RMSprop algorithm [65] for optimization. It must be noted that, the norm of the gradient of a $K$-Lipschitz Continuous function is at most $K$ at any point on that function. This implies that the function’s variations are bounded by a cone formed by lines with slopes $\pm K$. Enforcing $K$-Lipschitz Continuity on the Discriminator ensures that the Wasserstein loss is valid, continuous and differentiable which leads to training sta-
Chapter 5. Device Modeling: GANs

WGAN's training was further improved by enforcing the 1-Lipschitz Continuity constraint through a Gradient Penalty (GP) instead of weight clipping [68]. In this improved scheme, i.e., WGAN-GP, weight clipping and batch normalization are removed from the Discriminator and a regularizer, the gradient penalty, is added to the Discriminator's loss and ADAM algorithm [69] is used for optimization. For the purposes of this study, we have adopted the WGAN scheme for our Conditional GAN and the WGAN-GP scheme for the library of vanilla GANs.

5.3 Numerical Experiments

We implemented our models using TensorFlow. In the first phase of our experiments, we used the MNIST (Modified National Institute of Standards and Technology) handwritten digits dataset [70], which is a standard benchmark for computer vision tasks, to validate our architectures and implementation. This provides a visual way to verify that the code setup is working as intended. In other words, it provides a visual proof of convergence for the architecture. These experiments were carried out using an NVIDIA Tesla K40M GPU. It became clear through these experiments that WGAN-GP could not be satisfactorily extended to a conditional architecture, therefore we chose the scheme that worked best for each setting. The results are shown in Fig. 5.3. As can be seen from the figure, the generators of both the CGAN and the vanilla GAN can generate plausible examples from the dataset and the training metrics (generator loss, discriminator loss and the gradient penalty) are all stable and converge.

In the second phase, we trained our networks on LPFDA dataset (rfant 1, rfant 2, rfant 3) using a Desktop PC with Intel Core i7 Processor. Since our training examples are vectors that are formed by putting together various geometric parameters, each element of the vector (or each column of data in the 500 × 10 dataset) must be
considered a separate data channel and scaled with its own statistics (this is similar to scaling each of the Red, Green and Blue channels of a color image separately with its own statistics). To implement this scaling scheme, in a pre-processing step, each column of the dataset was normalized using its own minimum and maximum values (min-max feature scaling) and at inference time, network predictions were scaled back up using the same statistics. For the CGAN, the training time is approximately 10 minutes (for 125 epochs and a batch size of 128). For each vanilla GAN the training time is approximately 1.5 minutes (for 25 epochs and a batch size of 128); Therefore, training the entire library of vanilla GANs takes about 3 minutes. After training, we used each model to generate 50 predictions for each class. The total run time for each model is about 37 milliseconds (i.e., about 0.4 milliseconds for each generated example). We assessed the accuracy of the predicted structural parameters by simulating them in CST Microwave Studio and calculating the corresponding Q-factor values. Comparisons between the statistics of the predictions and those of the dataset along with the accuracy of the predictions are shown in Fig. 5.4. Box plots show that both models have demonstrated very good ability in learning the statistics of each parameter; the minimum and maximum of the predictions always fall within the interval defined by those of the dataset and there are significant similarities between the first quartile, median and the third quartile of the two. Scatter plots show that the CGAN outperforms the library of label-switched vanilla GANs. The superior performance of CGAN is expected since it is experimentally demonstrated that using side information, e.g., class labels, can significantly improve the quality of generated samples [71]. However, in both cases the accuracy of label 0 is lower than that of label 1; this uneven level of accuracy can be attributed to the much narrower Q-factor distribution of label 0 compared to that of label 1.
Chapter 5. Device Modeling: GANs

Figure 5.3: (a). Vanilla GAN (b). Conditional GAN

Figure 5.3: (a). Vanilla GAN (b). Conditional GAN
Chapter 5. Device Modeling: GANs

Figure 5.4: (a). library of Vanilla GANs (b). Conditional GAN
Chapter 6

Deep-Fake Signals: GANs

In the context of electromagnetic compatibility (EMC) compliance testing, we recognize that a realistic model of unintended emissions from the Device Under Test (DUT) is not only a function of its hardware design but also the state of the software that is running on it. As such, developing optimal EMC regulatory thresholds requires that the unintended emissions resulting from the state of the software running on the DUT be taken into account. Therefore, characterization of these emissions becomes essential. However, since there can be countless numbers of states, for both normal and anomalous operation of the software, characterization can’t be achieved by simply collecting a number of measurements. Therefore, we postulate that by training a generative model, specifically a GAN, on the available measurements, the model can learn their data-generating distribution and is able to produce new samples from it without any limitations. An example of the effects that imposing conservative regulatory thresholds can have is the delayed deployment of 5G in the vicinity of airports [72]. Since the ways that signals from 5G devices can interfere with older altimeter models are not fully understood, FAA demands complete filtering of 5G signals, resulting in large compliance costs and delays in deployment. Our research aims to provide solutions for such real-world problems.
To assess the performance of the proposed methodology, we consider the problem of unintended emissions from the motherboard of a PC when a substantial load is placed on its CPU, RAM or GPU. We propose to represent the measurements in each category by their spectra and then use those spectra to train a GAN for that category. In the inference phase, similar signals to measurements in a specific category can be generated by, first, asking the pertaining GAN to produce a number of spectra and, then, transform those GAN-generated spectra back into the time-domain. The main underlying assumption in this scheme is that the shared characteristic of the measurements in each category, i.e., the fact that they all represent intense activity in a certain component (or lack thereof in the case of the Idle category), can be quantitatively represented by the spectrum of frequencies that comprise the measured signals.

### 6.1 The Dataset

To collect our dataset, we put considerable load on a certain component and record the EM radiation emitted by the motherboard using a probe that is placed in its vicinity. Signals are sampled with $f_s = 20$ GHz in a span of 2 $\mu$s resulting in 40,000 samples. The signals in the Idle category were recorded when the computer was sitting idle, in other words, no major loads were placed on the CPU, RAM or the GPU components. The signals in the other categories (i.e., the CPU, RAM and GPU categories) were recorded when processes were running which put significant load on the pertaining components. Each category of the dataset contains one hundred of these recordings. Samples from each category are shown in Fig. 6.1(a).

We use Short-Time Fourier Transform (STFT) \cite{73, 74} to represent the measured signals in the frequency domain. The STFT process can be inversed to obtain a time-domain signal. However, to reconstruct the original time-domain signal, the
analysis window must satisfy the Constant Overlap-Add (COLA) constraint. This process of conversion to spectra and back into the original time-domain signal is illustrated in Fig. 6.1(b).
6.2 Numerical Experiments

Our Proposed scheme requires a mechanism for measuring the similarity of the examples in each category with themselves as well as measuring the similarity of the GAN-generated signals with those in the dataset. We propose to measure similarity using correlation. Specifically, in each category, we compute the cross-correlation between each signal and all other signals. For any two signals, computing the cross-correlation results in an array of values, each of which represents the cross-correlation between the two signals at a certain amount of lag. We choose the maximum value of the cross-correlation for each pair of signals. Furthermore, we adopt an ‘upper triangular’ approach to prevent calculating the same correlation multiple times. We start from one signal and compute its correlation with all other 99 signals (since each category has 100 signals). When moving to the next signal, we omit the first signal because the correlation between the second and the first is already computed. Therefore, using correlation as a measure of similarity, we can establish a baseline for membership in each category. These baseline plots, shown in Fig. 6.2, can then be used to decide whether or not GAN-generated signals are similar to the dataset.

As mentioned above, the main underlying assumption in our proposed scheme is that the shared characteristic of the measurements in each category can be quantitatively represented by the spectrum of frequencies that comprise the measured signals. To experimentally verify this assumption, we conduct a sanity-check experiment in which we transform the measurements, rearrange the order of the spectra and inverse transform them to obtain new signals. Samples from these new signals are shown in Fig. 6.3. Finally, we measure the similarity of these new signals with the ones in the dataset. For each category, the cross-correlation of each generated signal is computed with all the signals in the dataset and these values are plotted on top of the baseline plots as shown in Fig. 6.4. As can be seen from these plots, the new signals are similar to the dataset.
Figure 6.2: Baselines for membership in a category
Figure 6.3: New signals by rearranging spectra
Chapter 6. Deep-Fake Signals: GANs

Figure 6.4: Similarity of the new signals with the dataset. Cross-correlation values for the new signals are shown in red.

Figure 6.4: Similarity of the new signals with the dataset. Cross-correlation values for the new signals are shown in red.
Figure 6.5: Samples from the GAN-generated signals along with training metrics for each GAN.
Chapter 6. Deep-Fake Signals: GANs

Figure 6.6: Similarity of the GAN-generated signals with the dataset. Cross-correlation values for the GAN-generated signals are shown in red.

Figure 6.6: Similarity of the GAN-generated signals with the dataset. Cross-correlation values for the GAN-generated signals are shown in red.
Chapter 6. Deep-Fake Signals: GANs

With this experimental affirmation, we proceed to train a WGAN-GP for each category using the spectra of its signals. In the inference phase, GANs are asked to generate new spectra which are then inverse-transformed into time-domain to produce new signals. Samples from the GAN-generated signals are shown in Fig. 6.5. The similarity plots for these signals are shown in Fig. 6.6; From these plots, the GAN-generated signals appear to be similar to the dataset. However, experimental testing is needed before this can be decisively concluded.
Chapter 7

Conclusions

This work examined the various ways in which Deep Learning models can be utilized to assist the conventional engineering workflows.

In the field of computational EM, the FDTD method was cast as a video prediction task and various sequence processing models, such as LSTMs, Transformers and GNNs were experimented with to accomplish future frame prediction. The trained models were then used as building blocks in predicting the next frame in a DDM-based parallelization scheme. Future directions for this line of research could include constructing larger Transformers by modifying the current implementation to utilize multiple GPUs.

In the field of device design and development, GANs were utilized to realize a data-driven database of near-optimal designs. Future directions for this line of research could include expanding the vector of representation of the antennas to include radiation characteristics such as half power beam-width, operational bandwidth, etc. Finally, for the Deep-Fake signals generated by the GAN, experimental procedures must be devised to assess the similarity of the GAN-generated signals with the dataset.
Appendices
Appendix A

Derivations: ANNs
Table 1. Demonstration of the Back-Propagation Algorithm for a Three-Layered Network

Layer Three

\[
\frac{\partial E}{\partial \mathbf{W}^{(3)}} \equiv \left[ \begin{array}{ccc}
\frac{\partial E}{\partial W_{11}^{(3)}} & \frac{\partial E}{\partial W_{12}^{(3)}} & \frac{\partial E}{\partial W_{13}^{(3)}} \\
\frac{\partial E}{\partial W_{21}^{(3)}} & \frac{\partial E}{\partial W_{22}^{(3)}} & \frac{\partial E}{\partial W_{23}^{(3)}} \\
\frac{\partial E}{\partial W_{31}^{(3)}} & \frac{\partial E}{\partial W_{32}^{(3)}} & \frac{\partial E}{\partial W_{33}^{(3)}} \\
\frac{\partial E}{\partial W_{41}^{(3)}} & \frac{\partial E}{\partial W_{42}^{(3)}} & \frac{\partial E}{\partial W_{43}^{(3)}} \\
\end{array} \right]
\]

\[
= \left( f'(3) \circ \delta^{(3)} \right) \mathbf{a}^{(2)^T}
\]

\[
f'(3) \equiv \left[ \begin{array}{c}
\frac{\partial a_{1}^{(3)}}{\partial z_{1}^{(3)}} \\
\frac{\partial a_{2}^{(3)}}{\partial z_{2}^{(3)}} \\
\frac{\partial a_{3}^{(3)}}{\partial z_{3}^{(3)}} \\
\frac{\partial a_{4}^{(3)}}{\partial z_{4}^{(3)}}
\end{array} \right]
\]
\[
\delta^{(3)} \equiv \begin{bmatrix}
\frac{\partial E}{\partial a_1^{(3)}} \\
\frac{\partial E}{\partial a_2^{(3)}} \\
\frac{\partial E}{\partial a_3^{(3)}} \\
\frac{\partial E}{\partial a_4^{(3)}} 
\end{bmatrix}
\]

Layer Two

\[
\frac{\partial E}{\partial W^{(2)}} \equiv \begin{bmatrix}
\frac{\partial E}{\partial W_{11}^{(2)}} & \frac{\partial E}{\partial W_{12}^{(2)}} & \frac{\partial E}{\partial W_{13}^{(2)}} & \frac{\partial E}{\partial W_{14}^{(2)}} \\
\frac{\partial E}{\partial W_{21}^{(2)}} & \frac{\partial E}{\partial W_{22}^{(2)}} & \frac{\partial E}{\partial W_{23}^{(2)}} & \frac{\partial E}{\partial W_{24}^{(2)}} 
\end{bmatrix}
\]

\[
= \left( \frac{\partial E}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} a_1^{(1)} \right) \left( \frac{\partial E}{\partial a_2^{(2)}} \frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} a_2^{(1)} \right) \left( \frac{\partial E}{\partial a_3^{(2)}} \frac{\partial a_3^{(2)}}{\partial z_3^{(2)}} a_3^{(1)} \right) \left( \frac{\partial E}{\partial a_4^{(2)}} \frac{\partial a_4^{(2)}}{\partial z_4^{(2)}} a_4^{(1)} \right)
\]

\[
= (f^{(2)} \circ \delta^{(2)}) \bar{a}^{(1)T}
\]

\[
f^{(2)} \equiv \begin{bmatrix}
\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \\
\frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} 
\end{bmatrix}
\]

\[
\delta^{(2)} \equiv \begin{bmatrix}
\frac{\partial E}{\partial a_1^{(2)}} \\
\frac{\partial E}{\partial a_2^{(2)}} 
\end{bmatrix}
\]

\[
= \sum_{k=1}^{4} \frac{\partial E}{\partial a_k^{(3)}} \frac{\partial a_k^{(3)}}{\partial z_k^{(3)}} W_{k1}^{(3)}
\]

\[
= \sum_{k=1}^{4} \frac{\partial E}{\partial a_k^{(3)}} \frac{\partial a_k^{(3)}}{\partial z_k^{(3)}} W_{k2}^{(3)}
\]

\[
= W^{(3)T} (f^{(3)} \circ \delta^{(3)})
\]
Layer One

\[
\frac{\partial E}{\partial W^{(1)}} \equiv \begin{bmatrix}
\frac{\partial E}{\partial W_{11}^{(1)}} & \frac{\partial E}{\partial W_{12}^{(1)}} & \frac{\partial E}{\partial W_{13}^{(1)}} & \frac{\partial E}{\partial W_{14}^{(1)}} \\
\frac{\partial E}{\partial W_{21}^{(1)}} & \frac{\partial E}{\partial W_{22}^{(1)}} & \frac{\partial E}{\partial W_{23}^{(1)}} & \frac{\partial E}{\partial W_{24}^{(1)}} \\
\frac{\partial E}{\partial W_{31}^{(1)}} & \frac{\partial E}{\partial W_{32}^{(1)}} & \frac{\partial E}{\partial W_{33}^{(1)}} & \frac{\partial E}{\partial W_{34}^{(1)}}
\end{bmatrix}
\]

\[
= \left( f^{(1)} \odot \delta^{(1)} \right) \tilde{a}^{(0)^T}
\]

\[f^{(1)} \equiv \begin{bmatrix}
\frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \\
\frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \\
\frac{\partial a_3^{(1)}}{\partial z_3^{(1)}}
\end{bmatrix}
\]

\[\delta^{(1)} \equiv \begin{bmatrix}
\frac{\partial E}{\partial a_1^{(1)}} \\
\frac{\partial E}{\partial a_2^{(1)}} \\
\frac{\partial E}{\partial a_3^{(1)}}
\end{bmatrix}
\]

\[
= \sum_{k=1}^{2} \frac{\partial E}{\partial a_k^{(2)}} \frac{\partial a_k^{(2)}}{\partial z_k^{(2)}} W_k^{(2)}
\]

\[
= W^{(2)^T} \left( f^{(2)} \odot \delta^{(2)} \right)
\]
The Assumptions

The elements of the input/output are i.i.d (independent and identically distributed). (1-a)

The elements of the weight matrix are i.i.d (independent and identically distributed). The expected value of the elements of the weight matrix is assumed to be zero. (1-b)

The elements of the input/output and of the weight matrix are independent of each other. (1-c)

The elements of the $\delta$ are i.i.d (independent and identically distributed). (1-d)

The elements of the $\delta$ and of the weight matrix are independent of each other. (1-e)

The probability density function of the pre-non-linearity output, denoted by $p(z)$ is symmetric and centered at zero, i.e., it produces the probability of $\frac{1}{2}$ on both the positive and the negative parts of the $z$ axis. (1-f)

The Variance of the Multiplication of Two Independent RVs

$$Var[XY] = E[X^2Y^2] - E[XY]^2$$


The Law of the Unconscious Statistician

$$E[g(X)] = \int_{-\infty}^{+\infty} g(x)p(x) \, dx$$

The $a^{(l)}$

$$a^{(l)} = f(\overline{W}^{(l)} \overline{a}^{(l-1)})$$

The $k^{th}$ element of the $a^{(l)}$

$$a_k^{(l)} = f \left( \sum_{i=1}^{(0l-1)+1} w_{ki}^{(l)} a_i^{(l-1)} \right)$$
The $\delta^{(l)}$

$$\delta^{(l)} = W^{(l+1)^T} \left( f^{(l+1)} \odot \delta^{(l+1)} \right) \quad (3-a)$$

The $k^{th}$ element of the $\delta^{(l)}$

$$(3-a) \quad \rightarrow \quad \delta_k^{(l)} = \sum_{i=1}^{O_{l+1}} w_{l,k}^{(l+1)} \delta_i^{(l+1)} f_i^{(l+1)} \quad (3-b)$$

Weight Distribution for the ReLU Non-linearity

The $f(z)$

$$f(z) \equiv \max(0, z) \quad (4-a)$$

$$E[f^2(z)] = \int_{-\infty}^{+\infty} f^2(z) p(z) \, dz \quad (4-b)$$

$$(1-f), \quad (1-2), \quad (4-a) \quad \rightarrow \quad E[f^2(z)] = \frac{1}{2} E[z^2]$$

$$E[f'^2(z)] = \int_{-\infty}^{+\infty} f'^2(z) p(z) \, dz \quad (4-c)$$

$$= \frac{1}{2}$$

The Variance of $z_k^{(l)}$

Since the elements of $a^{(l)}$ and, by extension, $z^{(l)}$ are i.i.d, the $k^{th}$ element of $a^{(l)}$ and $z^{(l)}$ are denoted by $a_k^{(l)}$ and $z_k^{(l)}$.

$$(1-a) \quad \rightarrow \quad E[z^{(l)}] = \sum_{i=1}^{O_{l-1}+1} w_{k,i}^{(l)} a_i^{(l-1)} \quad (5-a)$$

$$(1-b), \quad (1-c), \quad (2-b), \quad (5-a) \quad \rightarrow \quad E[z^{(l)}] = \sum_{i=1}^{O_{l-1}+1} E[w_{k,i}^{(l)} a_i^{(l-1)}] \quad (5-b)$$

$$= \sum_{i=1}^{O_{l-1}+1} E[w_{k,i}^{(l)}] E[a_i^{(l-1)}]$$

$$= 0$$
\[ V \text{var}[z^{(l)}] = V \text{var} \left[ \sum_{i=1}^{O_{l-1}+1} w_{k,i}^{(l)} a_i^{(l-1)} \right] \]

\[ = \sum_{i=1}^{O_{l-1}+1} V \text{var} \left[ w_{k,i}^{(l)} a_i^{(l-1)} \right] \]

\[ = \sum_{i=1}^{O_{l-1}+1} E \left[ (w_{k,i}^{(l)})^2 \right] E \left[ (a_i^{(l-1)})^2 \right] - E^2 \left[ w_{k,i}^{(l)} a_i^{(l-1)} \right] \]

\[ = \sum_{i=1}^{O_{l-1}+1} E \left[ (w_{k,i}^{(l)})^2 \right] E \left[ (a_i^{(l-1)})^2 \right] - E^2 \left[ w_{k,i}^{(l)} E [a_i^{(l-1)}] \right] \]

\[ = \frac{1}{2} \sum_{i=1}^{O_{l-1}+1} V \text{var} \left[ w_{k,i}^{(l)} \right] E \left[ (a_i^{(l-1)})^2 \right] \]

\[ = \frac{1}{2} \sum_{i=1}^{O_{l-1}+1} V \text{var} \left[ w_{k,i}^{(l)} \right] V \text{var} \left[ z_i^{(l-1)} \right] \]

\[ = \frac{O_{l-1}}{2} \left( V \text{var} \left[ w_k^{(l)} \right] V \text{var} \left[ z^{(l-1)} \right] \right) \]

From (5-c) it is evident that, to have \( V \text{var}[z^{(l)}] = V \text{var}[z^{(l-1)}] \), the corresponding weight vector must satisfy \( \frac{O_{l-1}}{2} \left( V \text{var} \left[ w_k^{(l)} \right] \right) = 1 \). \hspace{1cm} (6)

**The Variance of \( \delta_k^{(l)} \)**

Since the elements of \( \delta^{(l)} \) are i.i.d., the \( k \)th element of \( \delta^{(l)} \) is denoted by \( \delta_k^{(l)} \) instead of \( \delta_k^{(l)} \). \hspace{1cm} (7-a)

\[ E[\delta^{(l)}] = E \left[ \sum_{i=1}^{O_{l-1}+1} w_{i,k}^{(l+1)} \delta_i^{(l+1)} f_i^{(l+1)} \right] \]

\[ = \sum_{i=1}^{O_{l+1}} E \left[ w_{i,k}^{(l+1)} \delta_i^{(l+1)} f_i^{(l+1)} \right] \]

\[ = \sum_{i=1}^{O_{l+1}} E \left[ w_{i,k}^{(l+1)} E \left[ \delta_i^{(l+1)} f_i^{(l+1)} \right] \right] \]

\[ = 0 \]
\[
\text{Var}[\delta^{(l)}] = \text{Var} \left[ \sum_{i=1}^{O_{l+1}} w_{i,k}^{(l+1)} \delta_i^{(l+1)} f_i^{(l+1)} \right] \\
(1-a), (1-b), (1-c), (I-1), \rightarrow \]
\[
= \sum_{i=1}^{O_{l+1}} \text{Var} \left[ w_{i,k}^{(l+1)} \delta_i^{(l+1)} f_i^{(l+1)} \right] \\
= \sum_{i=1}^{O_{l+1}} \text{Var} \left[ w_{i,k}^{(l+1)} \right] \text{Var} \left[ \delta_i^{(l+1)} f_i^{(l+1)} \right] \\
= \frac{1}{2} \sum_{i=1}^{O_{l+1}} \text{Var} \left[ w_{i,k}^{(l+1)} \right] \text{Var} \left[ \delta_i^{(l+1)} \right] \\
= \frac{O_{l+1}}{2} \text{Var} \left[ w_{k}^{(l+1)} \right] \text{Var} \left[ \delta^{(l+1)} \right] \\
(7-c) \rightarrow \\
\text{From } (7-c) \text{ it is evident that, to have } \text{Var}[\delta^{(l)}] = \text{Var}[\delta^{(l+1)}], \text{ the corresponding weight vector must satisfy } \frac{O_{l+1}}{2} \left( \text{Var} \left[ w_{k}^{(l+1)} \right] \right) = 1. \quad (8) \\
\text{This implies that for the } l^{\text{th}} \text{ layer, } \frac{O_{l}}{2} \left( \text{Var} \left[ w_{i}^{(l)} \right] \right) = 1.
\]

**The He Initialization**

To keep constant the variance of \( z_k^{(l)} \) as they propagate forward and the variance of \( \delta_k^{(l)} \) as they propagate backward through the layers, either (6) or (8) can be used to initialize the weights of the \( l^{\text{th}} \) layer.

**Normal Distribution**

\[ w^{(l)} \sim \mathcal{N} \left( 0, \frac{2}{O_{l-1}} \right) \]

(9) \rightarrow

**Uniform Distribution**

\[ w^{(l)} \sim U \left( -\frac{6}{O_{l-1}}, \frac{6}{O_{l-1}} \right) \]

**Weight Distribution for the Hyperbolic Tangent Non-linearity**

**The \( f(z) \)**

\[ f(z) \equiv \tanh(z) \quad (10-a) \]

**Tylor Expansion**

\[ \tanh(z) \approx z - \cdots \quad (10-b) \]
\[ E[f^2(z)] = \int f^2(z)p(z) \, dz \]
\[ = \int z^2p(z) \, dz \quad \text{(10-c)} \]
\[ = E[z^2] \]
\[ E[f'^2(z)] = \int (f'^2(z))p(z) \, dz \]
\[ = \int p(z) \, dz \quad \text{(10-d)} \]
\[ = 1 \]

**The Variance of \( z_k^{(l)} \)**

Since the elements of \( a^{(l)} \) and, by extension, \( z^{(l)} \) are i.i.d, the \( k^{th} \) element of \( a^{(l)} \) and \( z^{(l)} \) are denoted by \( a_k^{(l)} \) and \( z_k^{(l)} \) instead of \( a_k^{(l)} \) and \( z_k^{(l)} \).

\[ E[z^{(l)}] = E \left[ \sum_{i=1}^{(O_{l-1}+1)} w_{k,i}^{(l)} a_i^{(l-1)} \right] \quad \text{(11-a)} \]
\[ = \sum_{i=1}^{(O_{l-1}+1)} E \left[ w_{k,i}^{(l)} a_i^{(l-1)} \right] \quad \text{(11-b)} \]
\[ = \sum_{i=1}^{(O_{l-1}+1)} E \left[ w_{k,i}^{(l)} \right] E \left[ a_i^{(l-1)} \right] \]
\[ = 0 \]

\[ \text{Var}[z^{(l)}] = \text{Var} \left[ \sum_{i=1}^{(O_{l-1}+1)} w_{k,i}^{(l)} a_i^{(l-1)} \right] \quad \text{(11-c)} \]
\[ = \sum_{i=1}^{(O_{l-1}+1)} \text{Var} \left[ w_{k,i}^{(l)} a_i^{(l-1)} \right] \]
\[ = \sum_{i=1}^{(O_{l-1}+1)} E \left[ \left( w_{k,i}^{(l)} \right)^2 \left( a_i^{(l-1)} \right)^2 \right] - E^2 \left[ w_{k,i}^{(l)} a_i^{(l-1)} \right] \]
\[ = \sum_{i=1}^{(O_{l-1}+1)} E \left[ \left( w_{k,i}^{(l)} \right)^2 \right] E \left[ \left( a_i^{(l-1)} \right)^2 \right] - E^2 \left[ w_{k,i}^{(l)} \right] E^2 \left[ a_i^{(l-1)} \right] \]
\[ = \sum_{i=1}^{(O_{l-1}+1)} E \left[ \left( w_{k,i}^{(l)} \right)^2 \right] E \left[ \left( a_i^{(l-1)} \right)^2 \right] \]
\[
\begin{align*}
&= \sum_{i=1}^{O_{l+1}} \text{Var} \left[ w_{k,i}^{(l)} \right] E \left[ \left( x_i^{(l-1)} \right)^2 \right] \\
&= \sum_{i=1}^{O_{l+1}} \text{Var} \left[ w_{k,i}^{(l)} \right] E \left[ \left( z_i^{(l-1)} \right)^2 \right] \\
&= \sum_{i=1}^{O_{l+1}} \text{Var} \left[ w_{k,i}^{(l)} \right] \text{Var} \left[ z_i^{(l-1)} \right] \\
&= O_{l-1} \left( \text{Var} \left[ w_k^{(l)} \right] \text{Var} \left[ z^{(l-1)} \right] \right)
\end{align*}
\]

From (11-c) it is evident that, to have \( \text{Var} \left[ z^{(0)} \right] = \text{Var} \left[ z^{(l-1)} \right] \), the corresponding weight vector must satisfy \( O_{l-1} \left( \text{Var} \left[ w_k^{(l)} \right] \text{Var} \left[ z^{(l-1)} \right] \right) = 1. \) (12)

**The Variance of \( \delta_k^{(l)} \)**

Since the elements of \( \delta^{(l)} \) are i.i.d, the \( k^{th} \) element of \( \delta^{(l)} \) is denoted by \( \delta_k^{(l)} \) instead of \( \delta_k^{(l)} \).

\[
\begin{align*}
E[\delta^{(l)}] &= E \left[ \sum_{i=1}^{O_{l+1}} w_{i,k}^{(l+1)} \delta_i^{(l+1)} f_i^{(l+1)} \right] \\
&= \sum_{i=1}^{O_{l+1}} E \left[ w_{i,k}^{(l+1)} \delta_i^{(l+1)} f_i^{(l+1)} \right] \quad (13-a) \\
&= \sum_{i=1}^{O_{l+1}} E \left[ w_{i,k}^{(l+1)} \right] E \left[ \delta_i^{(l+1)} f_i^{(l+1)} \right] \\
&= 0 \\
\text{Var}[\delta^{(l)}] &= \text{Var} \left[ \sum_{i=1}^{O_{l+1}} w_{i,k}^{(l+1)} \delta_i^{(l+1)} f_i^{(l+1)} \right] \\
&= \sum_{i=1}^{O_{l+1}} \text{Var} \left[ w_{i,k}^{(l+1)} \delta_i^{(l+1)} f_i^{(l+1)} \right] \quad (13-c) \\
&= \sum_{i=1}^{O_{l+1}} \text{Var} \left[ w_{i,k}^{(l+1)} \right] \text{Var} \left[ \delta_i^{(l+1)} f_i^{(l+1)} \right] \\
&= O_{l+1} \text{Var} \left[ w_k^{(l+1)} \right] \text{Var} \left[ \delta^{(l+1)} \right]
\end{align*}
\]
From (13-c) it is evident that, to have $\text{Var}[\delta^{(l)}] = \text{Var}[\delta^{(l+1)}]$, the corresponding weight vector must satisfy $O_{l+1} \left( \text{Var} \left[ w_k^{(l+1)} \right] \right) = 1$. This implies that for the $l^{th}$ layer, $O_l \left( \text{Var} \left[ w_k^{(l)} \right] \right) = 1$. (14)

### The Glorot Initialization

To keep constant the variance of $a_k^{(l)}$ as they propagate forward and the variance of $\delta_k^{(l)}$ as they propagate backward through the layers, $\text{Var} \left[ w_k^{(l)} \right] = \frac{2}{O_l + O_{l-1}}$ can be used to initialize the weights of the $l^{th}$ layer. This satisfies both (12) and (14) when the $l^{th}$ layer is connected to layers of the same size at both its input and its output.

**Normal Distribution**

$$w^{(l)} \sim \mathcal{N} \left( 0, \frac{2}{O_l + O_{l-1}} \right)$$

**Uniform Distribution**

$$w^{(l)} \sim \mathcal{U} \left( -\sqrt{\frac{6}{O_l + O_{l-1}}}, \sqrt{\frac{6}{O_l + O_{l-1}}} \right)$$
Appendix B

Derivations: FDTD
Table 1. Derivation of the 2D TEz Update Equations

\[ \nabla \cdot \mathbf{D} = 0 \quad (1-a) \\
\nabla \cdot \mathbf{B} = 0 \quad (1-b) \\
\nabla \times \mathbf{E} = -\mu \frac{\partial \mathbf{H}}{\partial t} - \mathbf{M} \quad (1-c) \\
\nabla \times \mathbf{H} = \varepsilon \frac{\partial \mathbf{E}}{\partial t} + \mathbf{J} \quad (1-d) \\
\]

\[ M \equiv \sigma^m \mathbf{H} + \mathbf{M}_i \quad (2-a) \\
\mathbf{J} \equiv \sigma^e \mathbf{E} + \mathbf{J}_i \quad (2-b) \]

(1-c), (2-a) \quad \rightarrow \quad \nabla \times \mathbf{E} = -\mu \frac{\partial \mathbf{H}}{\partial t} - \sigma^m \mathbf{H} - \mathbf{M}_i \quad (3-a) \\
(1-d), (2-b) \quad \rightarrow \quad \nabla \times \mathbf{H} = \varepsilon \frac{\partial \mathbf{E}}{\partial t} + \sigma^e \mathbf{E} + \mathbf{J}_i \quad (3-b) \]

\[ \frac{\partial \mathbf{H}}{\partial t} = -\frac{1}{\mu} \nabla \times \mathbf{E} - \frac{\sigma^m}{\mu} \mathbf{H} - \frac{1}{\mu} \mathbf{M}_i \quad (4-a) \\
\frac{\partial \mathbf{E}}{\partial t} = \frac{1}{\varepsilon} \nabla \times \mathbf{H} - \frac{\sigma^e}{\varepsilon} \mathbf{E} - \frac{1}{\varepsilon} \mathbf{J}_i \quad (4-b) \]

\[ \frac{\partial H_x}{\partial t} = \left( \frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y} \right) - \frac{\sigma^m}{\mu} H_x - \frac{1}{\mu} M_{ix} \quad (5-a) \\
\frac{\partial H_y}{\partial t} = \left( \frac{\partial E_x}{\partial z} - \frac{\partial E_z}{\partial x} \right) - \frac{\sigma^m}{\mu} H_y - \frac{1}{\mu} M_{iy} \quad (5-b) \\
\frac{\partial H_z}{\partial t} = \left( \frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x} \right) - \frac{\sigma^m}{\mu} H_z - \frac{1}{\mu} M_{iz} \quad (5-c) \]

\[ \frac{\partial E_x}{\partial t} = \left( \frac{\partial H_y}{\partial z} - \frac{\partial H_z}{\partial y} \right) - \frac{\sigma^e}{\varepsilon} E_x - \frac{1}{\varepsilon} J_{ix} \quad (6-a) \\
\frac{\partial E_y}{\partial t} = \left( \frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x} \right) - \frac{\sigma^e}{\varepsilon} E_y - \frac{1}{\varepsilon} J_{iy} \quad (6-b) \\
\frac{\partial E_z}{\partial t} = \left( \frac{\partial H_x}{\partial y} - \frac{\partial H_y}{\partial x} \right) - \frac{\sigma^e}{\varepsilon} E_z - \frac{1}{\varepsilon} J_{iz} \quad (6-c) \]

2D TE\textsuperscript{z} Mode

\[ E_z = 0 \quad (7-a) \]
\[ \frac{\partial}{\partial z} = 0 \quad (7-b) \]

\[ \frac{\partial H_z}{\partial t} = \left( \frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x} \right) - \frac{\sigma^m}{\mu} H_z - \frac{1}{\mu} M_{iz} \quad (8-a) \\
\frac{\partial E_x}{\partial t} = \frac{\partial H_y}{\partial z} = \frac{\sigma^e}{\varepsilon} E_x - \frac{1}{\varepsilon} J_{ix} \quad (8-b) \]
\[
\frac{\partial E_y}{\partial t} = -\frac{1}{\varepsilon} \frac{\partial H_z}{\partial x} - \frac{\sigma^e}{\varepsilon} E_y - \frac{1}{\varepsilon} J_i y \tag{8-c}
\]

**Discretization**

**Discretization of Space**

The top-right corner of cell \((i, j)\) is located at \((i\Delta x, j\Delta y)\) \tag{9-a}

\(H_z(i, j)\) is located at \(\left((i - \frac{1}{2})\Delta x, (j - \frac{1}{2})\Delta y\right)\) \tag{9-b}

\(E_x(i, j)\) is located at \(\left((i - \frac{1}{2})\Delta x, (j - \frac{1}{2})\Delta y\right)\) \tag{9-c}

\(E_y(i, j)\) is located at \(\left((i - 1)\Delta x, \left(j - \frac{1}{2}\right)\Delta y\right)\) \tag{9-d}

**Discretization of \(H_z\)**

\[
\frac{\partial H_z}{\partial t} = \frac{1}{\mu} \left( \frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x} \right) - \frac{\sigma^m}{\mu} H_z - \frac{1}{\mu} M_{iz} \tag{10}
\]

Temporal Derivative of \(H_z\)

\[
\frac{\partial H_z}{\partial t} \approx \frac{1}{\Delta t} \left( H_z^{n+\frac{1}{2}}(i, j) - H_z^{n-\frac{1}{2}}(i, j) \right) \tag{11}
\]

Partial Derivative of \(E_x\) at the location of \(H_z\)

\[
\frac{\partial E_x}{\partial y} \approx \frac{1}{\Delta y} E_x^n \left((i - \frac{1}{2})\Delta x, (j - \frac{1}{2})\Delta y + \frac{1}{2}\Delta y\right) - \frac{1}{\Delta y} E_x^n \left((i - \frac{1}{2})\Delta x, (j - \frac{1}{2})\Delta y - \frac{1}{2}\Delta y\right) \tag{12}
\]

\[
\frac{\partial E_x}{\partial y} \approx \frac{1}{\Delta y} E_x^n \left((i - \frac{1}{2})\Delta x, (j + 1)\Delta y\right) - \frac{1}{\Delta y} E_x^n \left((i - \frac{1}{2})\Delta x, (j)\Delta y\right) \tag{13}
\]

\[
\frac{\partial E_x}{\partial y} \approx \frac{1}{\Delta y} \left( E_x^n (i, j + 1) - E_x^n (i, j) \right) \tag{14}
\]

Partial Derivative of \(E_y\) at the location of \(H_z\)

\[
\frac{\partial E_y}{\partial x} \approx \frac{1}{\Delta x} E_y^n \left((i - \frac{1}{2})\Delta x + \frac{1}{2}\Delta x, (j + \frac{1}{2})\Delta y\right) - \frac{1}{\Delta x} E_y^n \left((i - \frac{1}{2})\Delta x - \frac{1}{2}\Delta x, (j + \frac{1}{2})\Delta y\right) \tag{15}
\]
\[
\frac{\partial E_y}{\partial x} \approx \frac{1}{\Delta x} E_y^n \left( (i + 1) \Delta x + \frac{1}{2} \Delta x, (j - \frac{1}{2}) \Delta y \right) \\
- \frac{1}{\Delta x} E_y^n \left( (i - 1) \Delta x, (j - \frac{1}{2}) \Delta y \right)
\] (16)

\[
\frac{\partial E_y}{\partial x} \approx \frac{1}{\Delta x} \left( E_y^n (i + 1, j) - E_y^n (i, j) \right)
\] (17)

Value of \( H_z \)

\[
H_z \approx \frac{1}{2} \left( H_z^{n+\frac{1}{2}} (i, j) + H_z^{n-\frac{1}{2}} (i, j) \right)
\] (18)

\[
H_z^{n+\frac{1}{2}} (i, j) = \frac{2 \mu - \Delta t \sigma^m}{2 \mu + \Delta t \sigma^m} H_z^{n-\frac{1}{2}} (i, j)
\]

\[
+ \frac{2 \Delta t}{(2 \mu + \Delta t \sigma^m) \Delta y} \left( E_x^n (i, j + 1) - E_x^n (i, j) \right)
\]

\[
- \frac{2 \Delta t}{(2 \mu + \Delta t \sigma^m) \Delta x} \left( E_y^n (i + 1, j) - E_y^n (i, j) \right)
\]

\[
- \frac{2 \Delta t}{2 \mu + \Delta t \sigma^m} M_{iz}^n (i, j)
\]

Discretization of \( E_x \)

\[
\frac{\partial E_x}{\partial t} = \frac{1}{\varepsilon} \frac{\partial H_z}{\partial y} - \frac{\sigma^e}{\varepsilon} \frac{1}{\varepsilon} E_x - \frac{1}{\varepsilon} I_{ix}
\] (19)

Temporal Derivative of \( E_x \)

\[
\frac{\partial E_x}{\partial t} \approx \frac{1}{\Delta t} \left( E_x^{n+1} (i, j) - E_x^n (i, j) \right)
\] (20)

Partial Derivative of \( H_z \) at the location of \( E_x \)

\[
\frac{\partial H_z}{\partial y} \approx \frac{1}{\Delta y} \left( H_z^{n+\frac{1}{2}} \left( i - \frac{1}{2} \right) \Delta x, (j - 1) \Delta y + \frac{1}{2} \Delta y \right)
\]

\[
- \frac{1}{\Delta y} H_z^{n-\frac{1}{2}} \left( i - \frac{1}{2} \right) \Delta x, (j - 1) \Delta y - \frac{1}{2} \Delta y \right)
\] (21)

\[
\frac{\partial H_z}{\partial y} \approx \frac{1}{\Delta y} \left( H_z^{n+\frac{1}{2}} \left( i - \frac{1}{2} \right) \Delta x, (j - \frac{1}{2}) \Delta y \right)
\]

\[
- \frac{1}{\Delta y} H_z^{n-\frac{1}{2}} \left( i - \frac{1}{2} \right) \Delta x, \left( [j - 1] - \frac{1}{2} \right) \Delta y \right)
\] (22)

\[
\frac{\partial H_z}{\partial y} \approx \frac{1}{\Delta y} \left( H_z^{n+\frac{1}{2}} (i, j) - H_z^{n+\frac{1}{2}} (i, j - 1) \right)
\] (23)
Value of $E_x$

$$E_x \approx \frac{1}{2} \left( E_{x}^{n+1}(i,j) + E_{x}^{n}(i,j) \right)$$  \hspace{1cm} (23)

$$E_{x}^{n+1}(i,j) = \frac{2\varepsilon - \Delta t \sigma^{e}}{2\varepsilon + \Delta t \sigma^{e}} E_{x}^{n}(i,j)$$

$$+ \frac{2\Delta t}{2\varepsilon + \Delta t \sigma^{e}} \left( H_{z}^{n+\frac{1}{2}}(i,j) - H_{z}^{n+\frac{1}{2}}(i,j-1) \right)$$

$$- \frac{2\Delta t}{2\varepsilon + \Delta t \sigma^{e}} J_{ix}^{n+\frac{1}{2}}(i,j)$$

Discretization of $E_y$

$$\frac{\partial E_{y}}{\partial t} \approx - \frac{1}{\varepsilon} \frac{\partial H_{z}}{\partial x} - \frac{\sigma^{e}}{\varepsilon} E_{y} - \frac{1}{\varepsilon} J_{iy}$$  \hspace{1cm} (25)

Temporal Derivative of $E_y$

$$\frac{\partial E_{y}}{\partial t} \approx \frac{1}{\Delta t} \left( E_{y}^{n+1}(i,j) - E_{y}^{n}(i,j) \right)$$  \hspace{1cm} (26)

Partial Derivative of $H_z$ at the location of $E_y$

$$\frac{\partial H_{z}}{\partial x} \approx \frac{1}{\Delta x} H_{z}^{n+\frac{1}{2}} \left( (i-1)\Delta x + \frac{1}{2} \Delta x, (j-\frac{1}{2})\Delta y \right)$$

$$- \frac{1}{\Delta x} H_{z}^{n+\frac{1}{2}} \left( (i-1)\Delta x - \frac{1}{2} \Delta x, (j-\frac{1}{2})\Delta y \right)$$  \hspace{1cm} (27)

$$\frac{\partial H_{z}}{\partial x} \approx \frac{1}{\Delta x} H_{z}^{n+\frac{1}{2}} \left( [i-1] - \frac{1}{2} \Delta x, (j-\frac{1}{2})\Delta y \right)$$

$$- \frac{1}{\Delta x} H_{z}^{n+\frac{1}{2}} \left( [i-1] - \frac{1}{2} \Delta x, (j-\frac{1}{2})\Delta y \right)$$  \hspace{1cm} (28)

$$\frac{\partial H_{z}}{\partial x} \approx \frac{1}{\Delta x} \left( H_{z}^{n+\frac{1}{2}}(i,j) - H_{z}^{n+\frac{1}{2}}(i-1,j) \right)$$  \hspace{1cm} (29)

Value of $E_y$

$$E_y \approx \frac{1}{2} \left( E_{y}^{n+1}(i,j) + E_{y}^{n}(i,j) \right)$$  \hspace{1cm} (30)

$$E_{y}^{n+1}(i,j) = \frac{2\varepsilon - \Delta t \sigma^{e}}{2\varepsilon + \Delta t \sigma^{e}} E_{y}^{n}(i,j)$$

(25), (26), (29), (30) →
\[- \frac{2\Delta t}{(2\varepsilon + \Delta t\sigma e)\Delta x} \left( H_{z}^{n+\frac{1}{2}}(i,j) - H_{z}^{n+\frac{1}{2}}(i-1,j) \right) - \frac{2\Delta t}{2\varepsilon + \Delta t\sigma e} f_{ly}^{n+\frac{1}{2}}(i,j) \]

**Table 2. Derivation of the Split-PML Absorbing Boundary Condition**

<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>( \frac{\partial H_z}{\partial t} = \frac{1}{\mu} \left( \frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x} \right) - \frac{\sigma_m}{\mu} H_z ) [(1-a)]</td>
</tr>
<tr>
<td>(1)</td>
<td>( \frac{\partial E_x}{\partial t} = \frac{1}{\varepsilon} \frac{\partial H_z}{\partial y} - \frac{\sigma_e}{\varepsilon} E_x ) [(1-b)]</td>
</tr>
<tr>
<td>(1)</td>
<td>( \frac{\partial E_y}{\partial t} = -\frac{1}{\varepsilon} \frac{\partial H_z}{\partial x} - \frac{\sigma_e}{\varepsilon} E_y ) [(1-c)]</td>
</tr>
<tr>
<td>( H_z )</td>
<td>( \equiv 1 - \frac{\sigma_m}{\omega \mu} ) [(2-a)]</td>
</tr>
<tr>
<td>( E_x )</td>
<td>( \equiv 1 - \frac{\sigma_e}{\omega \varepsilon} ) [(2-b)]</td>
</tr>
<tr>
<td>( E_y )</td>
<td>( \equiv 1 - \frac{\sigma_x e}{\omega \varepsilon} ) [(2-c)]</td>
</tr>
</tbody>
</table>

**Splitting of \( H_z \) and Introduction of Anisotropy in Conductivities**

- \( H_z = H_{xz} + H_{zy} \) \[(3)\]
- \( (H_{xz}, \sigma_x^m), (H_{zy}, \sigma_y^m), (E_x, \sigma_x^e), (E_y, \sigma_y^e) \)

<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2)</td>
<td>( H_{xz} = j \frac{1}{\omega \mu S_x^m} \frac{\partial E_y}{\partial x} ) ( S_x^m \equiv 1 - j \frac{\sigma_x^m}{\omega \mu} ) [(4-a)]</td>
</tr>
<tr>
<td>(2)</td>
<td>( H_{zy} = -j \frac{1}{\omega \mu S_y^m} \frac{\partial E_x}{\partial y} ) ( S_y^m \equiv 1 - j \frac{\sigma_y^m}{\omega \mu} ) [(4-b)]</td>
</tr>
<tr>
<td>(2)</td>
<td>( E_x = -j \frac{1}{\omega \varepsilon S_x^e} \frac{\partial H_z}{\partial y} ) ( S_x^e \equiv 1 - j \frac{\sigma_x^e}{\omega \varepsilon} ) [(4-c)]</td>
</tr>
<tr>
<td>(2)</td>
<td>( E_y = j \frac{1}{\omega \varepsilon S_y^e} \frac{\partial H_z}{\partial x} ) ( S_y^e \equiv 1 - j \frac{\sigma_y^e}{\omega \varepsilon} ) [(4-d)]</td>
</tr>
</tbody>
</table>

**2D TE^2 Mode in Source-Free Region: Solutions**

<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E )</td>
<td>( \equiv (E_{0x} a_x + E_{0y} a_y) e^{-jk \cdot r} ) ( E_{0x}^2 + E_{0y}^2 \equiv E_0^2 ) [(5-a)]</td>
</tr>
<tr>
<td>( H )</td>
<td>( \equiv (H_{0x} a_z) e^{-jk \cdot r} ) ( = \left( \frac{1}{Z} E_0 a_z \right) e^{-jk \cdot r} ) [(5-b)]</td>
</tr>
</tbody>
</table>
\[ k \equiv k_x a_x + k_y a_y \] 

(5-c)

\[
(4-a), (5-a) \quad \rightarrow \quad H_{0zx} = \frac{k_x}{\omega \mu S_x^m} E_{0y} 
\]

(6-a)

\[
(4-b), (5-a) \quad \rightarrow \quad H_{0zy} = -\frac{k_y}{\omega \mu S_y^m} E_{0x} 
\]

(6-b)

\[
(4-c), (5-b) \quad \rightarrow \quad E_{0x} = -\frac{k_y}{\omega \varepsilon S_y^e} H_{0z} 
\]

(6-c)

\[
(4-d), (5-d) \quad \rightarrow \quad E_{0y} = \frac{k_x}{\omega \varepsilon S_x^e} H_{0z} 
\]

(6-d)

\[
(6-a), (6-d) \quad \rightarrow \quad H_{0zx} = \frac{k_x^2}{\omega^2 \mu S_x^m S_x^e} H_{0z} 
\]

(7-a)

\[
(6-b), (6-c) \quad \rightarrow \quad H_{0zy} = \frac{k_y^2}{\omega^2 \mu S_y^m S_y^e} H_{0z} 
\]

(7-b)

\[
(7) \quad \rightarrow \quad \omega^2 \mu \varepsilon = \left(\frac{k_x}{\sqrt{S_x^m S_x^e}}\right)^2 + \left(\frac{k_y}{\sqrt{S_y^m S_y^e}}\right)^2 
\]

(8)

\[
(8) \quad \rightarrow \quad k_x = \omega \sqrt{\mu \varepsilon} \sqrt{S_x^m S_x^e} \cos \theta 
\]

(9-a)

\[
(8) \quad \rightarrow \quad k_y = \omega \sqrt{\mu \varepsilon} \sqrt{S_y^m S_y^e} \sin \theta 
\]

(9-b)

\[
(5-a), (5-b), (6-c), (6-d), (9) \quad \rightarrow \quad Z \equiv \frac{E_0}{H_{0z}} = \frac{\mu W}{\varepsilon} 
\]

(10-a)

\[
W \equiv \sqrt{\left(\frac{S_y^m}{S_x^m} \cos \theta\right)^2 + \left(\frac{S_y^m}{S_y^e} \sin \theta\right)^2} 
\]

(10-b)

2D TE^2 Mode in Matched Medium

\[
(6-c), (6-d), (9-a), (9-b), (10) \quad \rightarrow \quad E = \frac{E_0}{W} \left( -\sqrt{\frac{S_y^m}{S_y^e}} \sin \theta a_x + \sqrt{\frac{S_y^m}{S_x^e}} \cos \theta a_y \right) e^{-jkr} 
\]

(11-a)

\[
(5-b) \quad \rightarrow \quad H = \left(\frac{E_0}{Z} a_x\right) e^{-jkr} 
\]

(11-b)

\[
(9) \quad \rightarrow \quad k = \omega \sqrt{\mu \varepsilon} \left(\sqrt{S_x^m S_x^e} \cos \theta a_x + \sqrt{S_y^m S_y^e} \sin \theta a_y \right) 
\]

(11-c)

\[
(10) \quad \rightarrow \quad Z = \frac{\mu W}{\varepsilon} \quad W \equiv \sqrt{\left(\frac{S_y^m}{S_x^m} \cos \theta\right)^2 + \left(\frac{S_y^m}{S_y^e} \sin \theta\right)^2} 
\]

(11-d)
Matching Conditions

\[ S_x^m = S_x^e \]  \hspace{1cm} (12-a)
\[ S_y^m = S_y^e \]  \hspace{1cm} (12-b)
\[ \sigma_x^m = \frac{\mu}{\varepsilon} \sigma_x^e \]  \hspace{1cm} (13-a)
\[ \sigma_y^m = \frac{\mu}{\varepsilon} \sigma_y^e \]  \hspace{1cm} (13-b)

\[ E = E_0 (-\sin \theta \ a_x + \cos \theta \ a_y) e^{-jkr} \]  \hspace{1cm} (13-c)
\[ H = \left( \frac{E_0}{Z} a_z \right) e^{-jkr} \]  \hspace{1cm} (13-d)

\[ k = \frac{\omega}{\sqrt{\mu \varepsilon}} (\cos \theta \ a_x + \sin \theta \ a_y) - j \frac{\sqrt{\mu}}{\sqrt{\varepsilon}} \left( \sigma_x^e \cos \theta \ a_x + \sigma_y^e \sin \theta \ a_y \right) \]  \hspace{1cm} (13-e)
\[ Z = \sqrt{\frac{\mu}{\varepsilon}} \]  \hspace{1cm} (13-f)

2D TE² Mode in Matched Medium: Reflection & Transmission

Incidence Field
\[ a_p^i = \cos \theta_i \ a_x + \sin \theta_i \ a_y \]  \hspace{1cm} (14-a)
\[ E^i = E_0 (-\sin \theta_i \ a_x + \cos \theta_i \ a_y) e^{-\sqrt{\frac{\mu}{\varepsilon}} \sigma_x^e \cos \theta_i \ a_x + \sigma_y^e \sin \theta_i \ a_y} e^{-j\omega \sqrt{\mu \varepsilon} a_p^i r} \]  \hspace{1cm} (14-b)
\[ H^i = \frac{1}{Z} (E_0 a_z) e^{-\sqrt{\frac{\mu}{\varepsilon}} \sigma_x^e \cos \theta_i \ a_x + \sigma_y^e \sin \theta_i \ a_y} e^{-j\omega \sqrt{\mu \varepsilon} a_p^i r} \]  \hspace{1cm} (14-c)

Reflected Field
\[ a_p^r = -\cos \theta_r \ a_x + \sin \theta_r \ a_y \]  \hspace{1cm} (15-a)
\[ E^r = E_0 (\sin \theta_r \ a_x + \cos \theta_r \ a_y) e^{-\sqrt{\frac{\mu}{\varepsilon}} \sigma_x^e \cos \theta_r \ a_x + \sigma_y^e \sin \theta_r \ a_y} e^{-j\omega \sqrt{\mu \varepsilon} a_p^r r} \]  \hspace{1cm} (15-b)
\[ H^r = \frac{1}{Z} (-\Gamma E_0 a_z) e^{-\sqrt{\frac{\mu}{\varepsilon}} \sigma_x^e \cos \theta_r \ a_x + \sigma_y^e \sin \theta_r \ a_y} e^{-j\omega \sqrt{\mu \varepsilon} a_p^r r} \]  \hspace{1cm} (15-c)

Transmitted Field
\[ a_p^t = \cos \theta_t \ a_x + \sin \theta_t \ a_y \]  \hspace{1cm} (16-a)
\[ E^t = TE_0 (-\sin \theta_t \ a_x + \cos \theta_t \ a_y) e^{-\sqrt{\frac{\mu}{\varepsilon}} \sigma_x^e \cos \theta_t \ a_x + \sigma_y^e \sin \theta_t \ a_y} e^{-j\omega \sqrt{\mu \varepsilon} a_p^t r} \]  \hspace{1cm} (16-b)
\[ H^t = \frac{1}{Z} (TE_0 a_z) e^{-\sqrt{\frac{\mu}{\varepsilon}} \sigma_x^e \cos \theta_t \ a_x + \sigma_y^e \sin \theta_t \ a_y} e^{-j\omega \sqrt{\mu \varepsilon} a_p^t r} \]  \hspace{1cm} (16-c)

Along the Boundary Between the Two Media
\[ r_b = y a_y \]  \hspace{1cm} (17-a)
\[ n = a_x \]  \hspace{1cm} (17-b)
\[ n \times (E_2 - E_1) = 0 \]  \hspace{1cm} (18-a)
\[ n \times (H_2 - H_1) = 0 \]  \hspace{1cm} (18-b)

\[ E_1 \equiv E^t \]  \hspace{1cm} (18-a)
\[ H_1 \equiv H^t \]  \hspace{1cm} (18-b)
\[(18) \rightarrow E_y^t(r_b) + E_y^r(r_b) = E_y^t(r_b) \quad (19-a)\]
\[H_z^t(r_b) + H_z^r(r_b) = H_z^t(r_b) \quad (19-b)\]

\[(14), (15), (16), \ldots \quad j)  
\quad + C_{hex}(i, j) (E_x^n \Delta F(i, j + 1) - E_x^n \Delta F(i, j))  
\quad + C_{hey}(i, j) (E_y^n \Delta F(i + 1, j) - E_y^n \Delta F(i, j))\]

\[E_0 \cos \theta_t e^{-j \sqrt{\mu \epsilon_1} \sin \theta_y y} e^{-j \omega \mu_0 \sin \theta_y y} \]
\[+ \Gamma E_0 \cos \theta_r e^{-j \sqrt{\mu \epsilon_1} \sin \theta_y y} e^{-j \omega \mu_0 \sin \theta_y y} \quad (20-a)\]
\[= TE_0 \cos \theta_t e^{-j \sqrt{\mu \epsilon_1} \sin \theta_y y} e^{-j \omega \mu_0 \sin \theta_y y} \]

\[E_0 \gamma_i \sin \theta y y \]
\[+ \Gamma E_0 \gamma_i \sin \theta y y \]
\[= TE_0 \gamma_i \sin \theta y y \]

\[\theta_t = \theta_r = \theta_i \]

With Condition $\sigma_{y1} = \sigma_{y2}$
\[1 + \Gamma = T \]
\[1 - \Gamma = T \]

**Table 3. Derivation of the TF/SF Consistency Equations**

Assume a TF/SF boundary on the Yee grid. Any field components on or inside the boundary is considered a *Total Field (TF)* and any field component outside the boundary is considered *Scattered Field (SF)*. Since $E^{Total} = E^{Scattered} + E^{inc}$ and $H^{Total} = H^{Scattered} + H^{inc}$, a correction term must be added to each update equation to enforce consistency.

**Region O1**

\[H_z^{n+1 \Delta F} (i, j) = C_{hh}(i, j)H_z^{n-1 \Delta F} (i, j) \]
\[= C_{hex}(i, j) \left( E_x^{n \Delta F} (i, j + 1) - E_x^{n \Delta F} (i, j) \right) \]
\[+ C_{hey}(i, j) \left( E_y^{n \Delta F} (i + 1, j) - E_y^{n \Delta F} (i, j) \right) \]

Update equations for $E_x$ and $E_y$ require no corrections because all their terms are in the SF region.

**Region O2**

\[H_z^{n+1 \Delta F} (i, j) = C_{hh}(i, j)H_z^{n-1 \Delta F} (i, j) \]
\[= C_{hex}(i, j) \left( E_x^{n \Delta F} (i, j + 1) - E_x^{n \Delta F} (i, j) \right) \]
\[+ C_{hey}(i, j) \left( E_y^{n \Delta F} (i + 1, j) - E_y^{n \Delta F} (i, j) \right) \]
\[
E_{y_n+1}^{TF}(i,j) = C_{eye}(i,j)E_{y_n}^{TF}(i,j) \\
+ C_{eyh}(i,j) \left( H_{z_{n+1/2}} (i,j) - H_{z_{n+1/2}} (i-1,j) \right)
\]

Update equation for \( E_x \) requires no corrections because all its terms are in the SF region.

**Region O3**

\[
H_{z_n+1/2}^{n+1/2SF}(i,j) = C_{hh}(i,j)H_{z_n}^{n-1SF} (i,j) \\
+ C_{hex}(i,j) \left( E_{x_n}^{nSF}(i, j+1) - E_{x_n}^{nSF}(i,j) \right) \\
+ C_{hey}(i,j) \left( E_{y_n}^{nSF}(i+1,j) - E_{y_n}^{nSF}(i,j) \right)
\]

\[
E_{x_n+1}^{TF}(i,j) = C_{exe}(i,j)E_{x_n}^{TF}(i,j) \\
+ C_{exh}(i,j) \left( H_{z_n+1/2}^{n+1/2SF} (i,j) - H_{z_n+1/2}^{n+1/2SF} (i-1,j) \right)
\]

Update equation for \( E_y \) requires no corrections because all its terms are in the SF region.

**Region O4**

\[
E_{x_n+1}^{TF}(i,j) = C_{exe}(i,j)E_{x_n}^{TF}(i,j) \\
+ C_{exh}(i,j) \left( H_{z_n+1/2}^{n+1/2SF} (i,j) - H_{z_n+1/2}^{n+1/2SF} (i-1,j) \right)
\]

Update equations for \( E_x \) and \( E_y \) require no corrections because all their terms are in the SF region.

**Region 11**

\[
E_{x_n+1}^{TF}(i,j) = C_{exe}(i,j)E_{x_n}^{TF}(i,j) \\
+ C_{exh}(i,j) \left( H_{z_n+1/2}^{n+1/2TF} (i,j) - H_{z_n+1/2}^{n+1/2TF} (i-1,j) \right)
\]

Update equations for \( H_z \) and \( E_y \) require no corrections because all their terms are in the TF region.

**Region 12**

Update equations for \( H_z \), \( E_x \) and \( E_y \) require no corrections because all their terms are in the TF region.
Region I3

Update equations for $H_z$, $E_x$ and $E_y$ require no corrections because all their terms are in the TF region.

Region I4

\[
E_{y}^{n+1TF}(i, j) = C_{ey}(i, j)E_{y}^{nTF}(i, j) \\
+ C_{eyh}(i, j)\left( H_{z}^{n+1TF}(i, j) - H_{z}^{n+1SF}(i - 1, j) \right)
\]

Update equations for $H_z$ and $E_x$ require no corrections because all their terms are in the TF region.
References


References


References


References


References


References


References


