Hybrid Model for Interpretable Time Series Analysis

A THESIS

Presented to the Department of Computer Science and Computer Engineering

California State University, Long Beach

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

Option in Computer Science

Committee Members:

Ju Cheol Moon, Ph.D

Roman Tankelevich, Ph.D

Seok-Chul Kwon, Ph.D

College Designee:

Hamid Rahai, Ph.D.

By Ruben Rosales

May 2020

Abstract

In this thesis, we study interpretable machine learning as applied to complex-valued time-series data. Scientists have studied the use of several machine learning methods such as Convolutional Neural Networks, Recurrent Neural Networks, and Support Vector Machines for time series classification. These methods, however, fall short of allowing users to visualize patterns within their dataset.

To address this issue, we propose an interpretable hybrid model that can be extended to any time series dataset. In the majority of existing work in the field of interpretability, black-box models tend to outperform white-box models consistently. However, instead of relying purely on one method, we propose a collaboration between the two. This allows for the performance of a black-box model while providing a more precise visualization of our dataset.

To accomplish this, we created a framework composed of two models, an ensemble method of classifiers that functions as a black-box model, and a white-box model that encodes time series as images. The white-box model attempts to classify them through a neural network and outputs all images correctly classified in both black-box and white-box models.

Acknowledgments

I want to thank Dr. Moon for his guidance and for allowing me to work on a project this extensive. I would also like to thank my friends and family for their continuous support throughout this thesis.

Contents

Abstract		ii
Acknowledgments		iii
Illustrations		v
Chapter 1	Introduction	1
Chapter 2	Data	3
Chapter 3	Related Work	4
Chapter 4	Preprocessing	5
Chapter 5	Models	11
Chapter 6	Markov Transition Fields	17
Chapter 7	Main Results	20
Chapter 8	Conclusion	23
Appendices		24
Appendix A	MTF Code	25
Appendix B	Hybrid Model Code	39

Illustrations

Figures

1.1 Overview of our proposed hybrid model. 1

2.1 Size of each dataset. 3

4.1 Example of Short-Time Fourier Transform. 7

4.2 Example of different types of wavelets [11]. 8

4.3 Example of a wavelet decomposition network. 10

5.1 Example of a Convolutional Neural Network [2]. 12

5.2 Example of a LSTM unit [5]. 13

5.3 CNN architecture used in our black-box model. 14

5.4 LSTM architecture used in our black-box model. 15

5.5 Overview of our black-box model. 16

6.1 Example of QxQ quantile bin matrix used to calculate MTF [18]. 17

6.2 Markov Transition Field where W_{ij} denotes the transition probability from quantile i to j. 18

6.3 CNN architecture for classifying images generated through MTF. 19

7.1 Results 20

7.2 Examples of images generated through MTF. 22

Tables

Chapter 1

Introduction

As time-series datasets become ever more popular, the need for deep learning models becomes that much greater. In our black-box model, we analyze deep learning models such as Convolutional Neural Networks, which have become increasingly popular and widely used in the last decade [4]. However, their complex nature results in a lack of understanding of its inner workings [17]. As the use and complexity of black-box models rises, so too does the need for an interpretable accessory. For this reason, we propose a hybrid model consisting of a black-box for classification and an interpretable white-box which highlights characteristics of a time series for users to understand.

The goal of our work is not to find an alternative to black-box models, but to provide a way for users to have an understanding of what features are prevalent in their dataset.

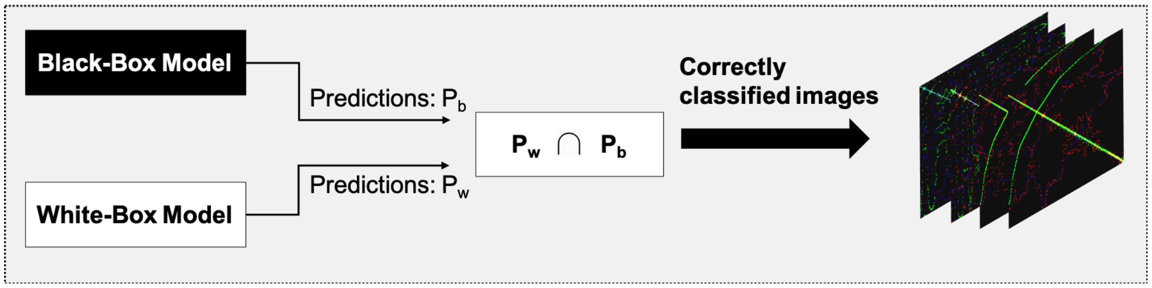


Figure 1.1: Overview of our proposed hybrid model.

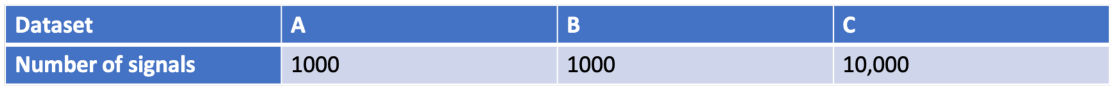
Contributions

We focus on creating deep learning architectures to classify our datasets as well as extending existing work to create our white-box model. Our final results show how robust our black-box model is with its competitive accuracy. Additionally, we show the effectiveness of our white-box model in understanding the features of a dataset not normally exposed through a black-box model.

Chapter 2

Data

We analyzed our model on three radio signal datasets, which we will refer to as dataset A, B, and C throughout this thesis. These datasets were generated through Spatial Modulation which is a transmission technique that uses multiple antennae which maps a block of information bits to two units, a symbol chosen from a constellation diagram, and a unique transmit antenna number that is chosen from a set of transmit antennae [12]. The method in which transmit antennae send and receive radio waves can be described through polarization. Two popular basis polarizations are horizontal linear, H, and vertical linear, V [14]. Our datasets contain data horizontally transmitted and vertically received (HV), as well as data vertically transmitted and vertically received (VV).



Dataset	A	B	C
Number of signals	1000	1000	10,000

Figure 2.1: Size of each dataset.

The two properties, HV and VV, are given to us in a raw and discrete format. Both properties consist of time and amplitude values but vary in length and in how they are processed. The raw signal consists of over 31,000 data points with the exact time it was received. The size of the discrete data can vary from 22 to 44 data points and is a processed representation of the raw signal that has equally spaced values. In this thesis, we focus on discrete data given that it is closer to datasets in real-world situations.

Chapter 3

Related Work

The study of interpretable hybrid models has been worked on extensively in the past few years as deep learning has become increasingly popular. Scientists take different approaches as to what makes a hybrid model interpretable and apply it to different domains, such as computer vision or time series. For example, in [16], they focus on creating a black-box model for the classification of time-series that outperforms similar architectures. However, their white-box model focuses on analyzing the importance of their model's layers. This helps users identify the importance of wavelet decomposition but not necessarily understand their dataset.

Additionally, other work we found tends to focus on non-time-series data or creating a general framework defining the trade-off between interpretability and black-box methods. For instance, [17] defines a set of rules to achieve a high level of classification with interpretability but does not give any competitive methods to achieve this.

In order to fill the gap for a high performing interpretable model for time-series classification, we propose our hybrid model, which focuses on extensibility and robustness. We achieve this through a combination of deep learning models for classification as well as a unique white-box method to explaining features within classes of a given dataset.

Chapter 4

Preprocessing

The radio signal data is presented in a complex-valued format that is unusable in a typical neural network, so in an attempt to make our model robust and reduce any complexity, we focused on using real values as features.

We tried numerous methods to extract real-valued features, including Fourier Transform, Short-Time Fourier Transform, a custom sliding window method, and polar coordinates taken directly from the signal's amplitude/phase value. Before applying any of these methods, we normalized all data using L2 normalization.

Transformations

Fourier Transform

The Fourier Transform (FT) is a mathematical tool that decomposes any function into a sum of sinusoidal basis functions. Each basis function is a complex value of a frequency, allowing us to view our data in the frequency domain as opposed to the amplitude domain.

One of the shortcomings of the FT is rooted in the Heisenberg Uncertainty Principle (HUP) [7]. The HUP states that the position and velocity of an object cannot be simultaneously measured, which can be applied to the time-frequency information of a signal. This means we cannot know which spectral components exist at any given time. The closest we can get is sampling at different ranges of time and finding a range of frequencies within that time frame. This method is described as the Short-Time Fourier Transform.

Short-Time Fourier Transform

The Short-Time Fourier transform (STFT) is a Fourier related transform used to determine the sinusoidal frequency and phase content of local sections of a signal as it changes over time [7]. Computing STFT requires the signal to be divided into segments of equal length and then have the FT applied to that segment [1]. This allows us to view the Fourier spectrum at a more granular level, which could potentially reveal different patterns amongst signals of different classes.

One of the issues with this method is that we can create very narrow window sizes, giving us a better understanding of the data with respect to time but losing an understanding of the frequency domain as a whole. Additionally, selecting an appropriate window size for segmenting the signal can be an arduous task that would require fine-tuning as well as increasing the dimensionality of a dataset.

Sliding Window Method

Similar to STFT, we segment our signal into N windows of equal length and perform FT on every segmentation. However, instead of keeping the output from FT, we only keep the mean value of each segmentation. This performed as well as STFT and gave us the advantage of reducing the dimensionality of our dataset.

Polar Coordinates

The representation of a complex number as a sum of a real and imaginary number, $z = x + iy$, is called its Cartesian representation. For every cartesian point, we calculate its radius and angular distance as real values and use those as features. This method outper-

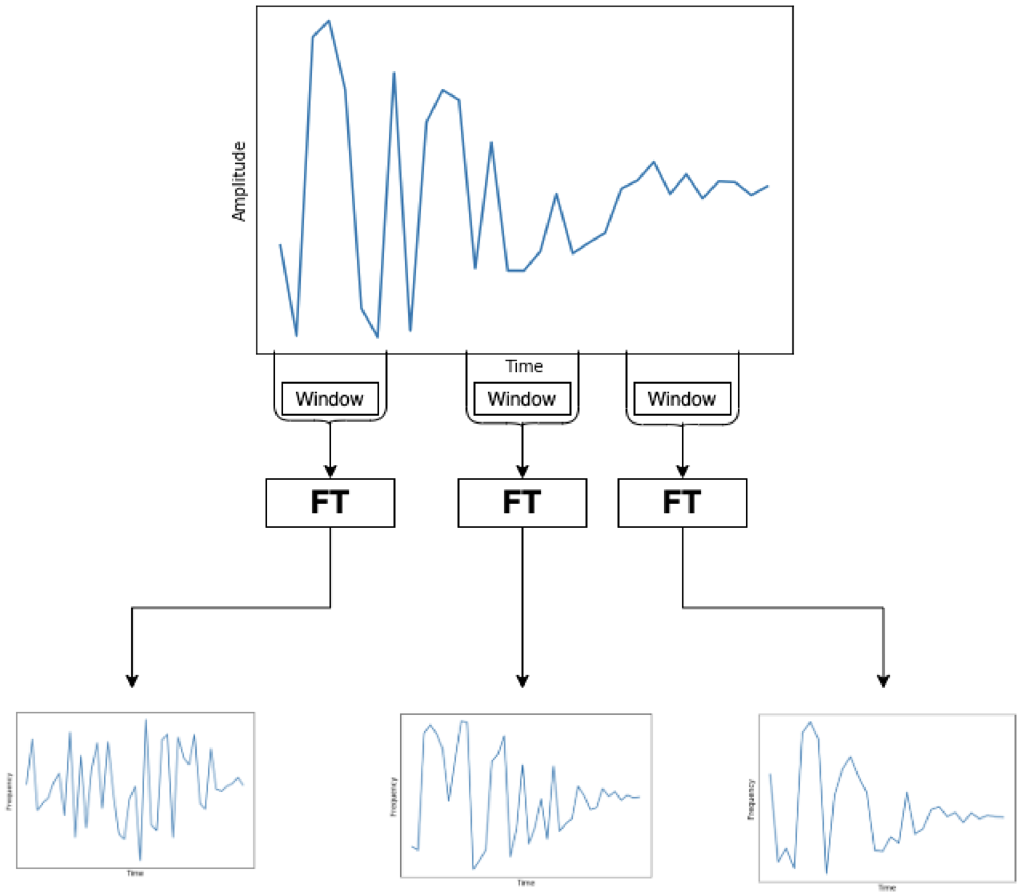


Figure 4.1: Example of Short-Time Fourier Transform.

formed the previous three methods but limited our feature size, which made it difficult to get consistent accuracy across all three of our datasets.

Wavelets

Wavelet transforms are similar to the FT in that they deconstruct a signal using representations of other signals [13]. The key difference is that wavelet signals are finite in time and frequency as opposed to sine and cosine signals, which can carry on indefinitely [15]. This allows us to extract information from a signal with respect to time and location.

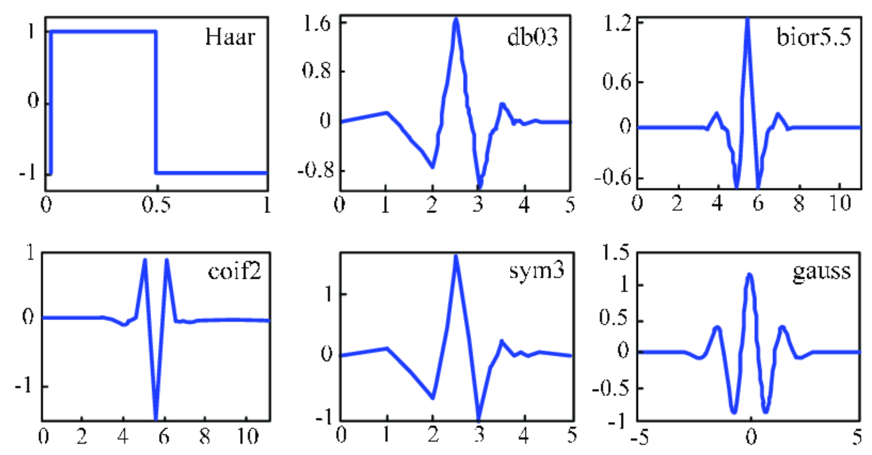


Figure 4.2: Example of different types of wavelets [11].

A signal is analyzed using different versions of dilated and translated basis functions called the mother wavelet [15]. There are two types of wavelet transformations, discrete and continuous [8]. In this thesis, we focus on discrete wavelet transformations, which use a discrete set of wavelet scales and translations to decompose the signal into a mutually orthogonal set of wavelets.

We take advantage of wavelets by applying discrete wavelet transforms as a filter-bank, meaning they are composed of cascading high-pass and low-pass filters. This allows us to split a signal into several frequency sub-bands [15]. We focus on wavelet decomposition as our feature extractor because it outperformed all other methods in terms of accuracy and training time.

Data Shape

A neural network requires all data to be the same shape. However, because we downsample in wavelet decomposition, our data size is halved, making it impossible to place all levels of decomposition into the same dataset. In order to circumvent this, we tested two different methods: resampling the data and treating each level of decomposition as a unique dataset.

Resampling

We use spline interpolation to resize each level of decomposition into a single array because it allows to retain properties of our original data even in a higher dimension [6]. We tested different sizes of interpolation from N to N/3, where N is the size of the largest discrete signal size, and found no reduction in performance.

Each Level of Decomposition as a Dataset

Figure 5.5 depicts our model in which each level of decomposition is a unique dataset. This method and the resampling method both achieved similar results, so we chose to move forward with this method.

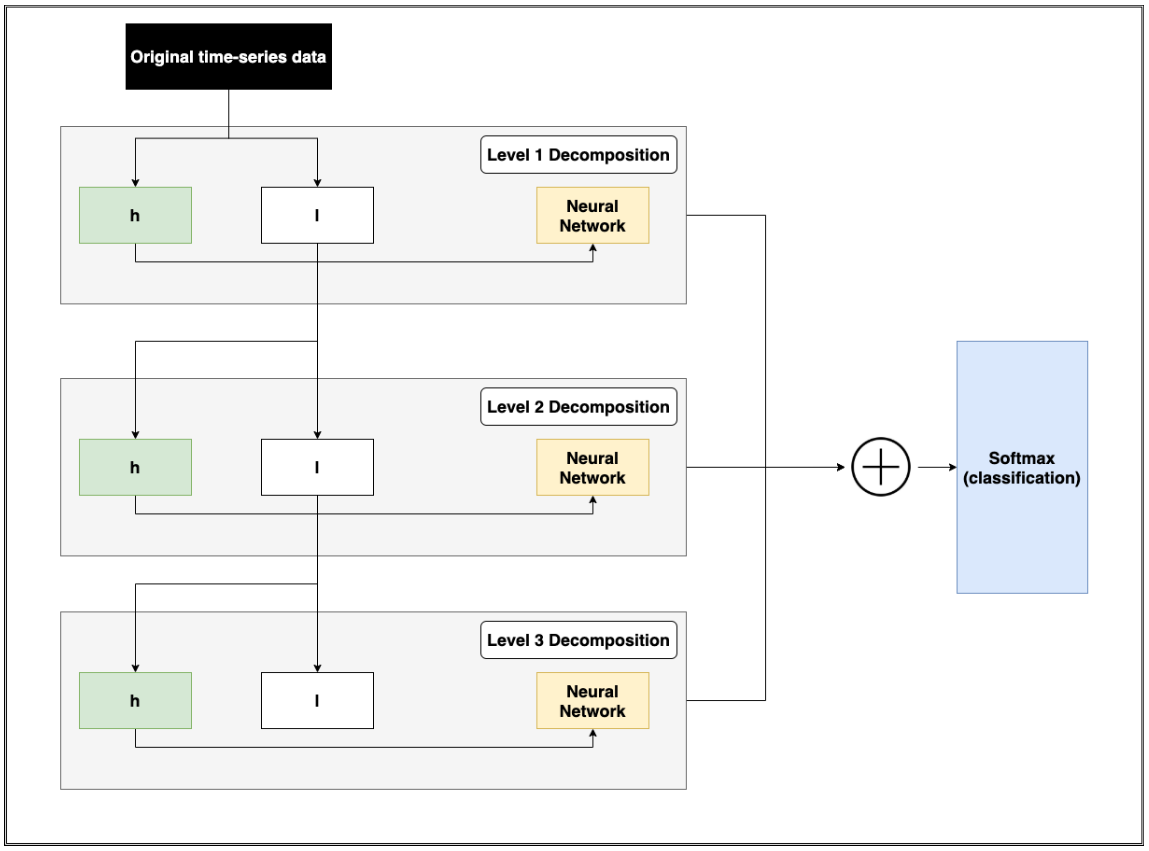


Figure 4.3: Example of a wavelet decomposition network.

Chapter 5

Models

Definition

Machine learning algorithms can be seen as learning a target function (f) that maps input data (X) to an output (Y). There are several techniques to make this work, but we focus on nonparametric algorithms, namely Convolutional Neural Networks (CNN) and Long Short-Term Memory networks (LSTM). Nonparametric algorithms attempt to make minimal assumptions about the form of the function to learn any form from data provided to it. An example would be a neural network as it has no prior knowledge of what it is classifying and attempts to generalize any new data points.

CNN

A Convolutional Neural Network is a type of deep neural network that can be applied to different domains such as computer vision or time series analysis. As seen in Figure X, it consists of an input and output layer as well as several hidden layers. A convolution is an operation between a vector of weights w against an input x [19]. It consists of taking the dot product between m and x in steps of a filter size defined by n.

Convolutional Neural Networks perform feature learning via non-linear transformations implemented as a series of layers [3]. The input data is a multidimensional array, called a tensor. The tensor is passed through an input layer, followed by a series of hidden layers to extract features, and an output layer, which in the case of classification, gives a probability for each class.

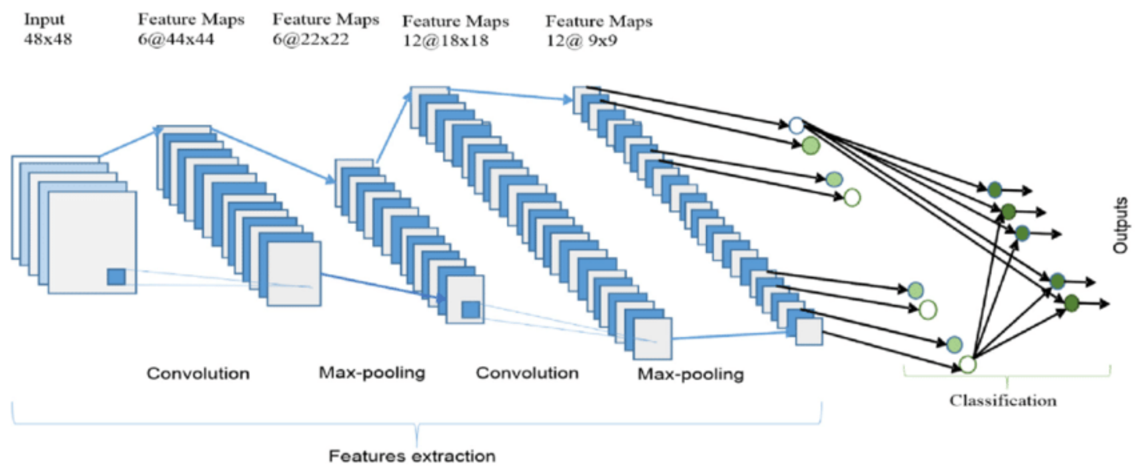


Figure 5.1: Example of a Convolutional Neural Network [2].

Hidden layers are crucial to neural networks because they help in determining which data representations are useful for explaining the relationships in the given data. Each layer consists of several kernels, which are feature detectors that convolve over the input data and output a transformed version of it.

LSTM

Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture used in the field of deep learning. Unlike standard feedforward neural networks, LSTM has feedback connections, processing not only single data points but also entire sequences of data.

A standard LSTM unit is composed of a cell, an input gate, an output gate, and a forget gate. The cell remembers values over arbitrary time intervals, and the three gates regulate the flow of information into and out of the cell [5].

LSTM networks are well-suited to classifying, processing, and making predictions based on time series data since there can be lags of unknown duration between significant events in a time series. LSTMs are developed to deal with the vanishing gradient problems that can are encountered when training traditional RNNs [5].

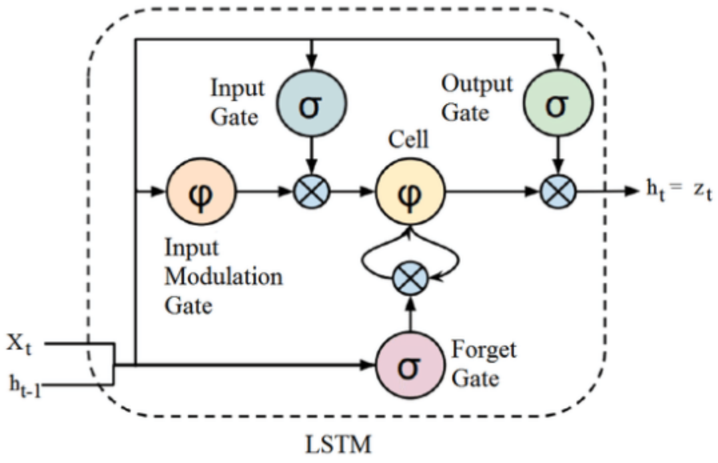


Figure 5.2: Example of a LSTM unit [5].

Multimodal Deep Learning

Due to the superior performance and computationally tractable representation capability in multiple domains such as visual, audio, and text, deep neural networks have gained tremendous popularity in multimodal learning tasks [9]. Typically, domain-specific neural networks are used on different modalities to generate their representations, and the individual representations are merged or aggregated [10]. Finally, the prediction is made on top of aggregated representation, usually with another neural network to capture the interactions between models and learn complex function mapping between input and output.

Architectures Used

CNN

Our CNN architecture consists of 3 convolutional layers each followed by a batch normalization and dropout layer then finally connected to a dense layer (Figure 5.3). We attempted to make it deeper but found inconsistent performance across all of our datasets.

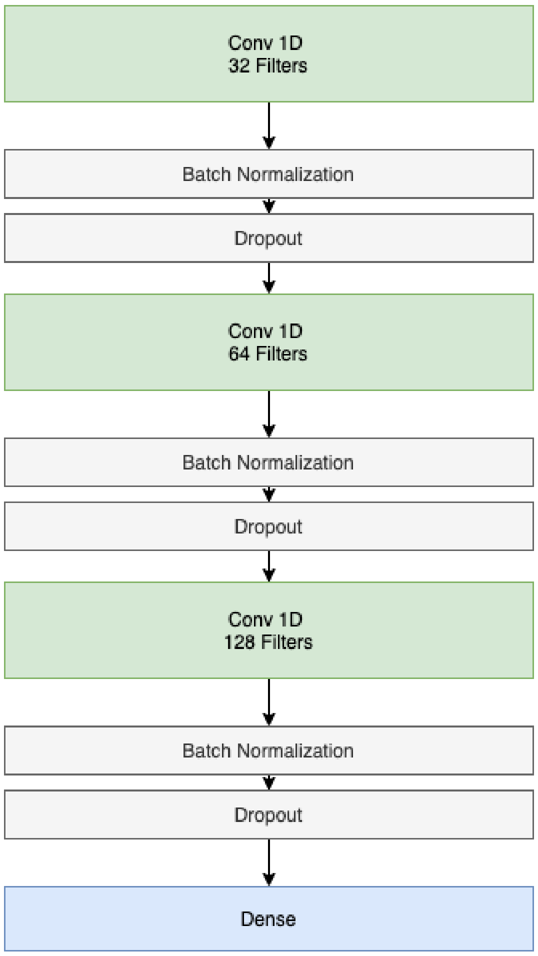


Figure 5.3: CNN architecture used in our black-box model.

LSTM

We wanted to keep our LSTM network as small as possible for simplicity, so we went with two LSTM layers of 128 hidden states and a dropout rate of 0.4 followed by a dense layer of 128 connections. We attempted different state sizes but found 128 to be the smallest number of units with the highest and most consistent rate of accuracy.

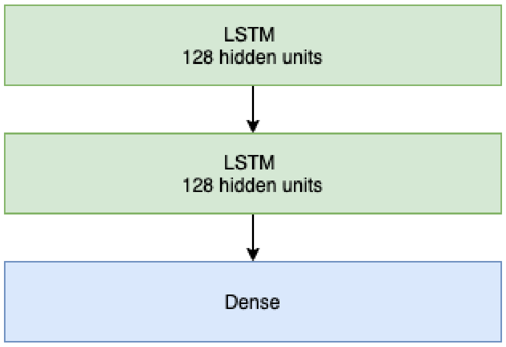


Figure 5.4: LSTM architecture used in our black-box model.

Final Model

We propose a multimodal learning architecture for time series classification, which is illustrated in Figure 5.5. The multimodal architecture allows our data to vary in size, and given that all three levels of decomposition have varying lengths due to downsampling, we propose that each model learn the representation of each distinct feature. For this model, we propose two architectures: a 1D CNN and an LSTM. We employ two CNN's and one LSTM, with each concatenated following their respective dense layer, after which classification is performed via a softmax layer.

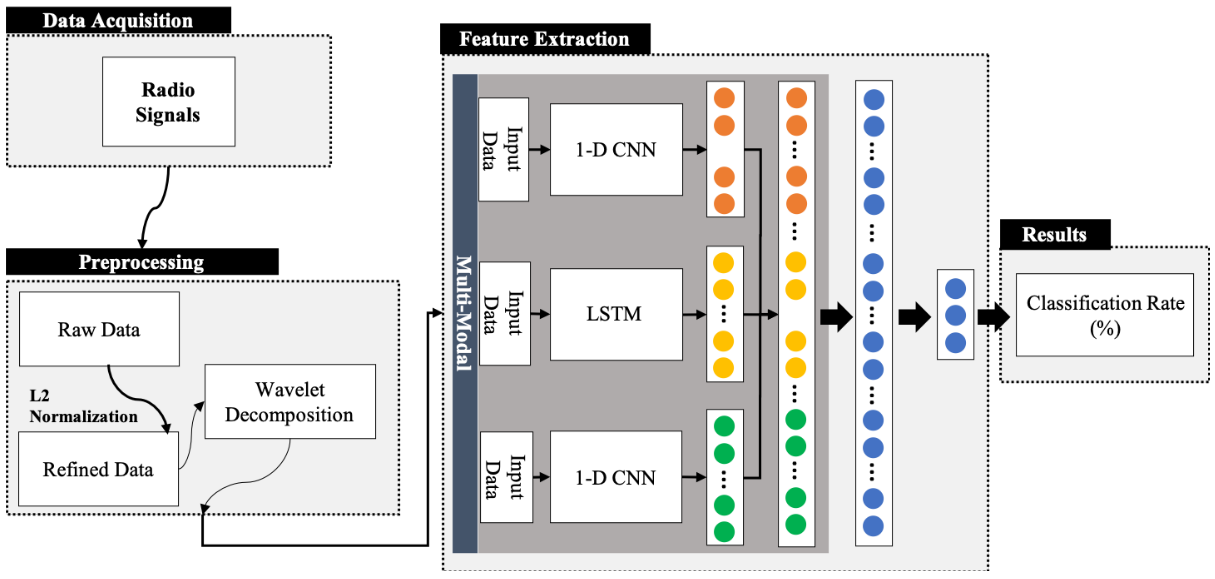


Figure 5.5: Overview of our black-box model.

Chapter 6

Markov Transition Fields

Method

We propose a framework similar to [18] for encoding dynamical transition statistics, but we continue their work by adding i$^{\text{th}}$ order Markov transition probabilities.

Given a time series X, we decompose its magnitude axis into three separate properties, P_1, P_2, P_3. We then identify the Q quantile bins for each property and assign each $x \in P_1, P_2, P_3$ to its corresponding bin q_j (j in [1, Q]). Thus, we construct three QxQ adjacency transition matrices, W_1, W_2, W_3, by counting transitions among quantile bins in the manner of an i$^{\text{th}}$ order Markov chain along the time axis.

$$\begin{array}{c c}
 & \begin{array}{cccc} A & B & C & D \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{bmatrix} 0.917 & 0.083 & 0 & 0 \\ 0.083 & 0.583 & 0.334 & 0 \\ 0 & 0.260 & 0.522 & 0.218 \\ 0 & 0.083 & 0.167 & 0.75 \end{bmatrix}
\end{array}$$

Figure 6.1: Example of QxQ quantile bin matrix used to calculate MTF [18].

W does not take into account the temporal axis, so to prevent any information loss, we construct a Markov Transition Field, M, for each W. The MTF denotes the probability of transitioning from q_i to q_j for each $x \in P$. This, in turn, allows us to consider the transition probability on the magnitude and temporal axis.

As described in [18] the MTF encodes the multi-span transition probabilities of the time series, but given that we have three different M's, we modify their approach and consider each M to be a separate color channel of RGB where M_1 is red, M_2 is blue, M_3

$$M = \begin{bmatrix} W_{ij}|x_1 \in q_i, x_1 \in q_j & \dots & W_{ij}|x_1 \in q_i, x_n \in q_j \\ \vdots & \ddots & \\ W_{ij}|x_n \in q_i, x_1 \in q_j & \dots & W_{ij}|x_n \in q_i, x_n \in q_j \end{bmatrix}$$

Figure 6.2: Markov Transition Field where W_{ij} denotes the transition probability from quantile i to j.

is green. Since each row in M is a probability from 0 to 1, we multiply it by 255 to get a color value.

Classification

We apply MTF to our dataset and classify our images using a network similar to AlexNet, as shown in Figure 6.3. We avoided using the CNN used in our black-box model in order to create a deeper and higher performing network. For this model, we define our input as images with dimensions 224x224, use a kernel size of 3x3, and apply max-pooling after every convolutional layer. We have five convolutional layers using 11, 256, 256, 384, and 384 filters, respectively, followed by two dense layers and a softmax layer.

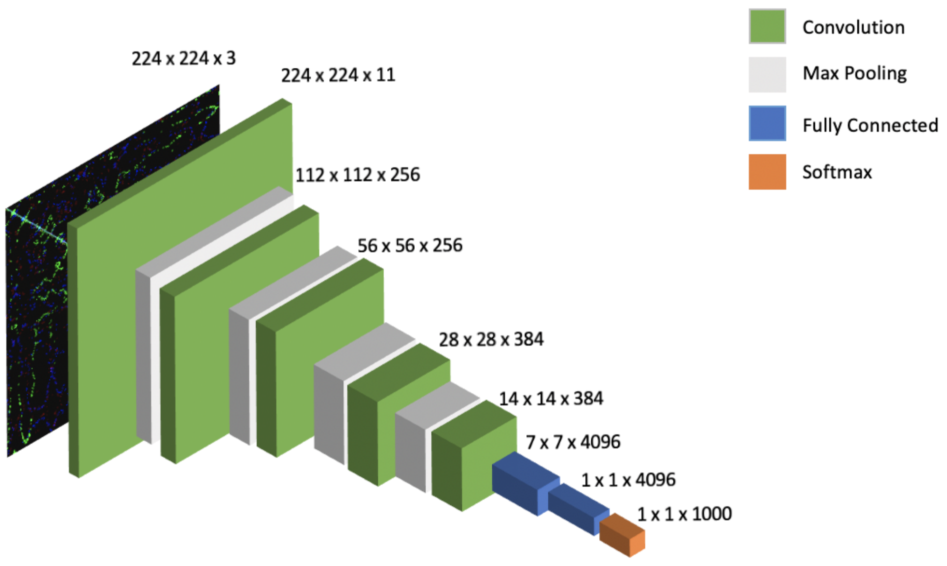


Figure 6.3: CNN architecture for classifying images generated through MTF.

Chapter 7

Main Results

Black-Box Model

Our black-box model is constructed of three different architectures: 1 LSTM and 2 CNN's that are all concatenated at their respective dense layer (Figure 5.5). The three features we use are the amplitude/phase of each level of decomposition for HV. We chose three because we wanted to increase the dimensionality of our data in the event that one had more descriptive features than the others.

Additionally, we tested our model using three levels of decomposition using the property VV as well as a combination of both VV and HV. We found no change in accuracy, therefore to limit the number of models we used, we only utilized the HV property.

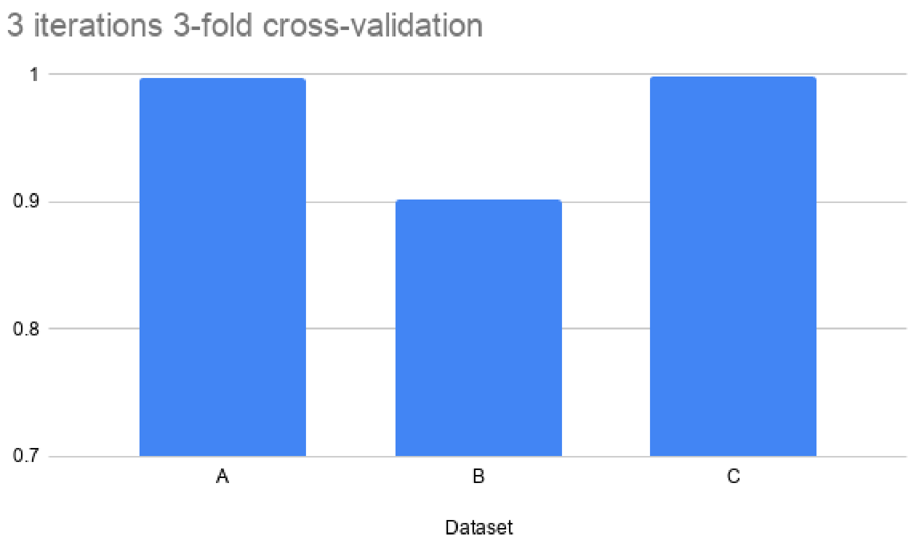


Figure 7.1: Results

To test our model, we split our data into training and test sets by randomly selecting 70% of our data as our training set and leaving the other 30% to be our test set. We then performed three-fold cross-validation and took the average of our results as the outcome.

Dataset C

Given that this dataset is ten times larget than our other two datasets, we wanted to try different training and test splits, going as low as 8% for training data and using the remaining 92% as test data. We ran the same number of cross-validations as our previous tests and found minimal difference in performance.

MTF

In order to generate images using MTF, we have to select three properties, and we chose P_1 as the radial value of HV's amplitude value, P_2 as the radial value of VV's amplitude value, and P_3 as the absolute squared distance between every P_1 and P_2.

We then split those images using a randomized 70/30 split where 70% of the data is our training set, and the other 30% is our test set. We were able to achieve 60% accuracy across all three of our datasets and show the features amongst the classes behave differently, as can be seen in Figure 7.2.

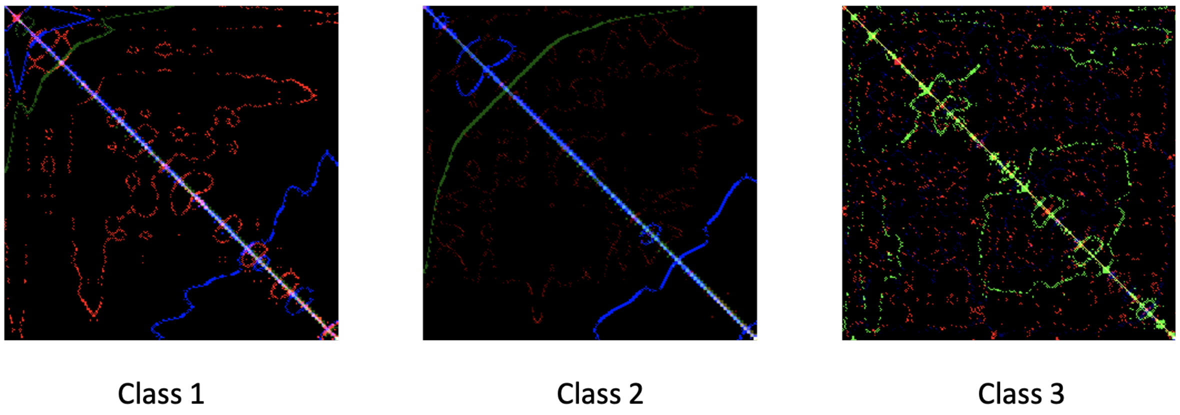


Figure 7.2: Examples of images generated through MTF.

Chapter 8

Conclusion

Throughout this thesis, we have stressed the importance of a hybrid machine learning algorithm as well as provided a framework that can successfully classify and visualize different classes when applied to time-series data.

We proposed an extension to existing work described in [18] to help users visualize their data while performing classification with 60% accuracy on that dataset and model. We also created a robust black-box model consisting of two different deep learning architectures that consistently provided competitive accuracy.

In the future, we would like to explore other methods for our white-box model that would act as an alternative to Markov Transition Fields in hopes that it produces higher accuracy than our current model. Additionally, we would like to extensively test our model on different time-series datasets outside of radio signal datasets.

Appendices

Appendix A

MTF Code

Main File

```
import util
import numpy as np
from complex_signal import ComplexSignal
import random
from scipy.io import loadmat
import os, glob
from multiprocessing import Pool
from keras.preprocessing.image import array_to_img
import scipy.ndimage as sc
from scipy.stats import entropy


def resize_(before, after_size):
    '''
    Function: resize_
    Purpose: Resizes data using spline interpolation
    '''
    before_size = before.shape[0]
    ratio = after_size / before_size
    return sc.zoom(before, ratio)
```

```
def scalar_wrapper(arg):
    '''
    Function: scalar_wrapper
    Purpose: Helper function to parallelize MTF
    '''

    arg.generateMTF()
    return arg


def main():

    #define parameters for MTF
    _IMAGE_SIZE = 300
    N = 300
    _CLASSES = 3
    _TIME_BINS = 100
    _THETA_BINS = 100
    _R_BINS = 100
    _ORDER = 1
    max_level = 1

    data = ['h_HV_SBR_1_discrete', 'h_HV_SBR_2_discrete', 'h_HV_SBR_3_discrete']
    data_vv = ['h_VV_SBR_1_discrete', 'h_VV_SBR_2_discrete', 'h_VV_SBR_3_discrete']
    path_name = "./dataset"
```

```
for _file in glob.glob(path_name):

  _data = []

  filename = _file
  file_contents = loadmat(filename)

  data = np.asarray(data)
  _CLASSES = data.shape[0]
  i = 0

  for _class in range(len(data)):
      r, theta, time = [], [], []

      #Read HV and VV property from data and normalize using L2 norm
      _data_hv = file_contents[data[_class]][:,1]
      _data_vv = file_contents[data_vv[_class]][:,1]
      _data_hv /= np.linalg.norm(_data_hv)
      _data_vv /= np.linalg.norm(_data_vv)

      #Define features that will have MTF applied to them
      r = np.abs(_data_hv)
      theta = np.abs(_data_vv)
```

```
    time = ( np.abs(_data_hv) − np.abs(_data_vv) ) **2

    #Resize data to allow for larger images to be created
    r = resize_(np.abs(r), N)
    theta = resize_(np.abs(theta), N)
    time = resize_(np.abs(time), N)

    time, theta, r = util.processRawMatrix(time, theta, r)

    ts = ComplexSignal(time, theta, r, _TIME_BINS, _THETA_BINS, _R_BINS, _IMAGE_SIZE, _ORDER, i

    _data.append(ts)

    i = i + 1 % _CLASSES
    if i == 0:
      i = _CLASSES

#use multiprocessing library to run all MTF for all classes of an individual signal in parallel
p = Pool(6)
res = p.map( scalar_wrapper, _data)
p.close()
p.join()

#Save images
```

```
        for item in res:
            pred_img = array_to_img(item.mtf)
            pred_img.save('../mtf/images/{0}/{1}_{2}.png'.format(item._class, filename, item._class))


if __name__== "__main__":
    main()
```

Calculating MTF

```
import numpy as np


class ComplexSignal:
    """
    Class: ComplexSignal
    Purpose: Process data with MTF
    """
    def __init__(self, time, theta, r, time_bins = 4, theta_bins = 4, r_bins = 4, image_size = 8, n_order = 4, _class=None)
        n_samples, n_features = time.shape
        self.image_size = image_size
        self.window_size = n_features // image_size
        self.time_series = [time, theta, r]
        self.num_bins = [time_bins, theta_bins, r_bins]
        self.sample = n_samples
        self.features = n_features
        self.mtf = None
    self.remainder = n_features % image_size
    self.n_order = n_order
    self._class = _class

def _mtf(self, binned_ts, num_bins):
    MTM = np.zeros((num_bins, num_bins))

    for i in range(self.n_order):
        temp_MTM = np.zeros((num_bins, num_bins))

        lagged_ts = np.vstack([binned_ts[:-1 - i], binned_ts[1 + i:]])

        np.add.at(temp_MTM, tuple(map(tuple, lagged_ts)), 1)
        temp_MTM *= (.4 ** (i+1))
        MTM += temp_MTM

    non_zero_rows = np.where(MTM.sum(axis=1) != 0)[0]
    MTM = np.multiply(MTM[non_zero_rows][:, non_zero_rows].T,
                      np.sum(MTM[non_zero_rows], axis=1)**(-1)).T

    MTF = np.zeros((self.features, self.features))
    list_values = [np.where(binned_ts == q) for q in non_zero_rows]

    for i in range(non_zero_rows.size):
            for j in range(non_zero_rows.size):
                MTF[tuple(np.meshgrid(list_values[i], list_values[j]))] = MTM[i, j]
    remainder = self.remainder
    if remainder == 0:
        return np.reshape(MTF,
                          (self.image_size, self.window_size,
                           self.image_size, self.window_size)
                          ).mean(axis=(1, 3))
    else:
        self.window_size += 1
        start, end, _ = segmentation(MTF.shape[0], self.window_size,
                                     False, self.image_size)
        AMTF = np.zeros((self.image_size, self.image_size))
        for i in range(self.image_size):
            for j in range(self.image_size):
                AMTF[i, j] = MTF[start[i]:end[i], start[j]:end[j]].mean()

        return AMTF


def generateMTF(self):
    ts, num_bins = self.time_series, self.num_bins
    mtf = []
    _bit_size = ( 2**16 ) - 1
    for idx in range(len(ts)):
            temp_mtf = None
            bins = self.createQuantiles(ts[idx], num_bins[idx])
            binned_ts = self.valuesToQuantiles(ts[idx], bins)
            temp_mtf = np.apply_along_axis(self._mtf, 1, binned_ts, num_bins[idx])[0]
            interp = np.interp(temp_mtf, (temp_mtf.min(), temp_mtf.max()), (0, _bit_size)).astype(np.uint8)
            mtf.append(interp)

        self.mtf = np.dstack((mtf[1], mtf[2], mtf[0]))
    def createQuantiles(self, data, num_bins):
        return np.percentile(data,
                             np.linspace(0, 100, num_bins + 1)[1:-1],
                             axis=1)

    def valuesToQuantiles(self, data, data_bins):
        n_samples, n_features = data.shape
        mask = np.r_[
            ~np.isclose(0, np.diff(data_bins, axis=0), rtol=0, atol=1e-8),
            np.full((1, n_samples), True)
        ]
        if isinstance(data_bins[0][0], complex):
            binned_ts = []
            data_bins = data_bins.flatten()
            for time_point in data[0]:
                binned = False
            for idx in range(len(data_bins)):
                # print(time_point, data_bins[idx])
                if abs(time_point) <= abs(data_bins[idx]):
                    binned_ts.append(idx)
                    binned = True
            if not binned:
                binned_ts.append(len(data_bins))
        return np.asarray(binned_ts).reshape(1, −1)
    else:
            return np.array([np.digitize(data[i], data_bins[:, i][mask[:, i]])
                                                   for i in range(n_samples)])
def segmentation(ts_size, window_size, overlapping, n_segments=None):
    """Compute the indices for Piecewise Agrgegate Approximation.


    Parameters
    −−−−−−−−−−
    ts_size : int
        The size of the time series.


    window_size : int
        The size of the window.


    overlapping : bool
        If True, overlapping windows may be used. If False, non−overlapping
    are used.

n_segments : int or None (default = None)
    The number of windows. If None, the number is automatically
    computed using ‘window_size‘.

Returns
−−−−−−−
start : array
    The lower bound for each window.

end : array
    The upper bound for each window.

size : int
    The size of ‘start‘.

”””
if n_segments is None:
    quotient = ts_size // window_size
    remainder = ts_size % window_size
    n_segments = quotient if remainder == 0 else quotient + 1

bounds = np.linspace(0, ts_size,
                n_segments + 1, endpoint=True).astype('int64')


start = bounds[:-1]
end = bounds[1:]
size = start.size


if not overlapping:
    return start, end, size
else:
    correction = window_size - end + start
    half_size = size // 2
    new_start = start.copy()
    new_start[half_size:] = start[half_size:] - correction[half_size:]
    new_end = end.copy()
    new_end[:half_size] = end[:half_size] + correction[:half_size]
    return new_start, new_end, size
```

Helper functions

```
import random
import math
import numpy as np


def polar(z):
    a= z.real
    b= z.imag

    r = math.hypot(a,b)

    x = a / (math.sqrt(a**2 + b **2))

    theta = math.acos(x)

    return r,theta


def normalize(a):

    return (a − np.min(a)) / (np.max(a) − np.min(a))


def generateTimeSeries(N, i):

    complex_numbers_time_series = []

    time = 1

    lower_bound = int(1 * i)

    upper_bound = int(10* i)

    for _ in range(N):

        rand_complex_number = complex(random.randint(lower_bound, upper_bound), random.randint(lower_bou

        time = random.randint(int(time) + 1, int(time) + 10) + (10 − random.uniform(0., 10.))

        time = time + (10 − random.uniform(0., 10.))


        complex_numbers_time_series.append((time, rand_complex_number))


    return complex_numbers_time_series
```

```
def processRawMatrix(theta, time, r):
    theta = np.asarray(normalize(theta)).reshape(1, −1)
    r = np.asarray(normalize(r)).reshape(1, −1)
    time = np.asarray(time).reshape(1, −1)

    return theta, time, r

def printHelper(theta, r, time, quantiles):
    for i in range(len(theta)):
        for j in range(len(theta[i])):
            print("Data Point:", theta[i][j], r[i][j], time[i][j])
            print("Quantile p,q,r:", quantile_bins[j])

def printPolar(theta, r):
    theta *= 2 * np.pi
    ax = plt.subplot(projection='polar')
    ax.set_rticks(r_bins.flatten()) # set radial ticks based on quantiles

    ax.plot(theta, r, 'k.')

    # ax.set_rticks([0.5, 1, 1.5, 2]) # less radial ticks
    ax.set_rlabel_position(22.5) # get radial labels away from plotted line
    ax.grid(True)
```

```
# ax.set_title(”Not scaled yet”, va=’bottom’)

plt.show()
```

Appendix B

Hybrid Model Code

Main File

```
import numpy as np
import tensorflow as tf


from black_box import blackBoxModel
from white_box import whiteBoxModel


from os import listdir
from os.path import isfile, join


def getListOfClassifiedImages(matched_predictions):
    '''
    Function: getListOfClassifiedImages
    Purpose: Given indices of images, return a list of names corresponding to those indices
    '''
    test_data_path = './cnn_data/test'
    image_files = [f for f in listdir(test_data_path) if isfile(join(test_data_path, f))]


    selected_files = []
    for i in range(len(image_files)):
        if i in matched_predictions:
            selected_files.append(image_files[i])


        return selected_files
def main():
    '''
    Function: main
    Purpose: Call white−box & black−box model and get intersection of correctly classified signals
    '''


    signals_dataset_path = "../Channel_Impulse_Response"
    image_dataset_path = "../data_whitebox"


    test_results_black_box = blackBoxModel(signals_dataset_path)
    test_results_white_box = whiteBoxModel(image_dataset_path)


    # get index of predicted class
    black_box_indexes = tf.argmax(test_results_black_box, axis=1)
    black_box_indexes = black_box_indexes.eval()
    white_box_indexes = tf.argmax(test_results_white_box, axis=1)
    white_box_indexes = white_box_indexes.eval()


    #getting the intersection of correctly classified items from black−box and white−box model
    matched_predictions = np.where(white_box_indexes == black_box_indexes)
```

```
        images = getListOfClassifiedImages(matched_predictions)


if __name__== "__main__":
    main()
```

White-Box Model

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout, Flatten, Conv2D, MaxPooling2D
from keras.layers.normalization import BatchNormalization
import numpy as np



def whiteBoxModel(image_dataset_path):
        '''
        function: whiteBoxModel
        Purpose: Main function that generates CNN model and evaluates on image dataset
        '''

        #define paramaters
        training_data_size = 490
        val_data_size = 210
        batch_size = 32
        epochs = 50
```

```
train_data_path = image_dataset_path + '/train'

val_data_path = image_dataset_path + '/val'

test_data_path = image_dataset_path + '/test'

img_size=224


model = getCNN()

model.compile(loss=keras.losses.categorical_crossentropy, optimizer='adam', metrics=['accuracy'])


generator = ImageDataGenerator(

    rotation_range=20,

    width_shift_range=0.2,

    height_shift_range=0.2,

    horizontal_flip=True,

    preprocessing_function=keras.applications.inception_v3.preprocess_input,

    validation_split=0.3,

)


train_gen = generator.flow_from_directory(

    train_data_path,

    batch_size=batch_size,

    subset='training',

    class_mode='categorical',

    target_size=(img_size, img_size)

)
```

```
val_gen = generator.flow_from_directory(
    val_data_path,
    batch_size=batch_size,
    subset='validation',
    class_mode='categorical',
    target_size=(img_size,img_size)
)

model.fit_generator(
    generator=train_gen,
    steps_per_epoch= training_data_size//batch_size ,
    epochs= epochs,
    validation_data=val_gen,
    validation_steps= val_data_size // batch_size
)

datagen = ImageDataGenerator()

generator = datagen.flow_from_directory(
            test_data_path,
            target_size=(img_size, img_size),
            batch_size=batch_size,
            class_mode='categorical',
            shuffle=True
        )

    return model.predict_generator(generator)

def getCNN():
    '''
    function: getCNN
    Purpose: Create 2d cnn for classifying images
    '''

    model = Sequential()

    model.add(Conv2D(filters=96, input_shape=(224,224,3), kernel_size=(11,11), strides=(4,4), padding="valid"))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2), padding="valid"))

    model.add(Conv2D(filters=256, kernel_size=(11,11), strides=(1,1), padding="valid"))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2), padding="valid"))

    model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding="valid"))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2), padding="valid"))
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding="valid"))
model.add(Activation('relu'))
 model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2), padding="valid"))

model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding="valid"))
model.add(Activation('relu'))

model.add(Flatten())

model.add(Dense(4096, input_shape=(224*224*3,)))
model.add(Activation('relu'))
model.add(Dropout(0.4))

model.add(Dense(4096))
model.add(Activation('relu'))
model.add(Dropout(0.4))

model.add(Dense(1000))
model.add(Activation('relu'))
model.add(Dropout(0.4))

model.add(Dense(3))
model.add(Activation('softmax'))
```

```
    return model
```

Black-Box Model

```
from keras.models import Sequential

from keras.layers.normalization import BatchNormalization

from keras.layers.convolutional import Conv1D, MaxPooling1D

from keras.layers.core import Activation

from keras.layers.core import Flatten

from keras.layers.core import Dropout

from keras.layers import Input, Dense, Concatenate

from keras.layers import LSTM

from keras.callbacks import CSVLogger

from keras.layers import merge


from keras.layers.core import Dense

from keras import backend as K

from keras.optimizers import Adam

from keras.models import Model

from sklearn.model_selection import StratifiedKFold

from keras.utils import to_categorical

import os

from keras.callbacks import ModelCheckpoint
```

```
import os, glob, shutil
from scipy import signal
from pywt import wavedec
import pywt

import scipy.ndimage

def resize_(before, after_size=300):
    '''
    Function: resize_
    Purpose: Resizes data using spline interpolation
    '''
    before_size = before.shape[0]
    ratio = after_size / before_size
    return scipy.ndimage.zoom(before, ratio)

def processData(data1, data2, data3, train_split = .7, random=True):
    '''
        Function: processData
        Purpose: Returns training and test data with the option of having them randomized
    '''

    #Generate array with numbers 0 − size of training data
    #then shuffle the array
    random_range = np.arange(data1.shape[0])

    if random:

        np.random.shuffle(random_range)


    labels = np.asarray([0,1,2]∗ (data1.shape[0] // 3))

    train_range = int(random_range.shape[0] ∗ train_split)


    x_train = data1[random_range[:train_range]]

    x_train1 = data2[random_range[:train_range]]

    x_train2 = data3[random_range[:train_range]]


    y_train = labels[random_range[:train_range]]


    x_test = data1[random_range[train_range:]]

    x_test1 = data2[random_range[train_range:]]

    x_test2 = data3[random_range[train_range:]]


    y_test = labels[random_range[train_range:]]


    shuffleImageData()

    return x_train, x_train1, x_train2, y_train, x_test, x_test1, x_test2, y_test


def shuffleImageData(random_range, train_range):
    '''
    function: shuffleImageData

    Purpose: Split data into training/val/test

    '''


    image_path = './images_full'


    image_files = [f for f in listdir(test_data_path) if isfile(join(test_data_path, f))]


    train_split_range = int(train_range * .7)


    train_set = image_files[random_range[:train_split_range]]

    val_set = image_files[random_range[train_split_range:train_range]]

    test_set = image_files[random_range[train_range:]]


    image_dataset_path = "../data_whitebox"


    copyData(image_path, image_dataset_path + '/train', train_set)

    copyData(image_path, image_dataset_path + '/val', val_set)

    copyData(image_path, image_dataset_path + '/test', test_set)


def copyData(current_dir, new_dir, files)

    '''

    function: copyData

    Purpose: Given a list of files and new directory, copy files into that directory
    '''

    for file in files:

        name = os.path.join(current_dir, file)

        if os.path.isfile( name ) :

            shutil.copy( name, new_dir)



def getCombinedModel(input_shape_1, input_shape_2, input_shape_3, num_classes):
    '''

    function: getCNN

    Purpose: Create model composed of 2 1−D CNN's and 1 LSTM

    '''


    kernel_size = 3


    input1 = Conv1D(32, (kernel_size), activation = "relu")(input_shape_1)

    input1 = BatchNormalization()(input1)

    input1 = Conv1D(filters = 64, kernel_size = (kernel_size),padding = 'Same', activation ='relu')(input1)

    input1 = BatchNormalization()(input1)


    input1 = Conv1D(filters = 128, kernel_size = (kernel_size),padding = 'Same', activation ='relu')(input1)

    input1 = BatchNormalization()(input1)

    input1 = Flatten()(input1)

    input1 = Dense(256, activation='relu')(input1)
```

```
input2 = Conv1D(32, (kernel_size), activation = ”relu”)(input_shape_2)

input2 = BatchNormalization()(input2)

input2 = Conv1D(filters = 64, kernel_size = (kernel_size),padding = ’Same’, activation =’relu’)(input2)

input2 = BatchNormalization()(input2)


input2 = Conv1D(filters = 128, kernel_size = (kernel_size),padding = ’Same’, activation =’relu’)(input2)

input2 = BatchNormalization()(input2)

input2 = Flatten()(input2)

input2 = Dense(256, activation=’relu’)(input2)



input3 = Conv1D(32, (kernel_size), activation = ”relu”)(input_shape_3)

input3 = BatchNormalization()(input3)

input3 = Conv1D(filters = 64, kernel_size = (kernel_size),padding = ’Same’, activation =’relu’)(input3)

input3 = BatchNormalization()(input3)


input3 = Conv1D(filters = 128, kernel_size = (kernel_size),padding = ’Same’, activation =’relu’)(input3)

input3 = BatchNormalization()(input3)

input3 = Flatten()(input3)

input3 = Dense(256, activation=’relu’)(input3)


encode_combined = Concatenate()([input1, input2, input3])

FC1 = Dropout(0.2)(encode_combined)
```

```
    predictions = Dense(num_classes, activation='softmax')(FC1)


    return Model(inputs=[input_shape_1,input_shape_2, input_shape_3], outputs=[predictions])



def generateBlackBoxData(path):
    '''
    function: generateBlackBoxData
    Purpose: Process signal data bu performing wavelet decomposition on it. Then, returning data already split int
    '''


    max_lev = 3


    class_names = ['h_HV_SBR_1_discrete', 'h_HV_SBR_2_discrete', 'h_HV_SBR_3_discrete']
    class_names_vv = ['h_VV_SBR_1_discrete', 'h_VV_SBR_2_discrete', 'h_VV_SBR_3_discrete']


    data1=[]
    data2=[]
    data3 = []
    num_classes = 3


    for _file in glob.glob(path):
        filename = _file
        file_contents = loadmat(filename)
```

```
for idx in range(num_classes):

    fc = file_contents[class_names[idx]]

    fc_vv = file_contents[class_names_vv[idx]]


    #normalize data

    data = fc / np.linalg.norm(fc)

    data_vv = fc_vv /np.linalg.norm(fc_vv)


    #Assign data to a variable so we can overwrite it after every level of decomposition

    hv = data[:22]

    vv = data_vv[:22]


    for level in range(0, max_lev):


        cA_hv, cD_hv = pywt.dwt(hv, 'db4', axis=0)

        cA1_vv, cD_vv = pywt.dwt(vv, 'db4', axis=0)


        hv = cD_hv

        vv = cD_vv


        if level == 0:

            data1.append(np.abs(cA_hv))
            if level == 1:
                data2.append(np.abs(cA_hv))

            if level == 2:
                data3.append(np.abs(cA_hv))

    data1 = np.asarray(data1)
    data2 = np.asarray(data2)
    data3 = np.asarray(data3)

    train_test_split = .7

    return processData(data1, data2, data3, train_test_split)

def blackBoxModel(dataset_path):
    '''
    function: blackBoxModel
    Purpose: Main function that proceses signal data and returns model with predictions
    '''

    batch_size = 8
    epochs = 20
    verbose = 0
    iterations = 3
```

```
n_fold = 3


dataset_path = dataset_path + '/*.mat'


x_train, x_train_1, x_train_2, y_train, x_test, x_test1, x_test2, y_test = generateBlackBoxData(dataset_path)


num_classes = to_categorical(y_train).shape[1]


input_shape = Input(shape=(x_train.shape[1], x_train.shape[2]))
input_shape_ = Input(shape=(x_train_1.shape[1], x_train_1.shape[2]))
input_shape1 = Input(shape=(x_train_2.shape[1], x_train_2.shape[2]))


for _ in range(n_fold):
skf = StratifiedKFold( n_splits=n_fold, shuffle=True)
for train, val in skf.split(x_train, y_train):
    model = getCombinedModel(input_shape, input_shape_, input_shape1, num_classes)
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

    history = model.fit(
                    [x_train[train], x_train_vv[train], x_train_angle_hv[train]],
                    to_categorical(y_train[train]),
                    batch_size=batch_size,
                    shuffle=True,
                    epochs=epochs,
                    verbose=verbose,
                    validation_data = ([x_train[val],
                        x_train_vv[val], x_train_angle_hv[val]],
                    to_categorical(y_train[val]))
                )
    del model

return model.predict([x_test, x_test1, x_test2], y_test)
```

Bibliography

[1] J.B. Allen and L.R. Rabiner. A unified approach to short-time Fourier analysis and synthesis. *Proceedings of the IEEE*, 65(11):1558–1564, November 1977. Conference Name: Proceedings of the IEEE.

[2] Md. Zahangir Alom, Tarek Taha, Chris Yakopcic, Stefan Westberg, Paheding Sidike, Mst Nasrin, Mahmudul Hasan, Brian Essen, Abdul Awwal, and Vijayan Asari. A State-of-the-Art Survey on Deep Learning Theory and Architectures. *Electronics*, 8:292, March 2019.

[3] Muhammad Aqib, Rashid Mehmood, Ahmed Alzahrani, Iyad Katib, Aiiad Albeshri, and Saleh Altowaijri. Smarter Traffic Prediction Using Big Data, In-Memory Computing, Deep Learning and GPUs. *Sensors*, 19:2206, May 2019.

[4] Zhicheng Cui, Wenlin Chen, and Yixin Chen. Multi-Scale Convolutional Neural Networks for Time Series Classification. *arXiv:1603.06995 [cs]*, May 2016. arXiv: 1603.06995.

[5] Jeff Donahue, Lisa Anne Hendricks, Marcus Rohrbach, Subhashini Venugopalan, Sergio Guadarrama, Kate Saenko, and Trevor Darrell. Long-term Recurrent Convolutional Networks for Visual Recognition and Description. *arXiv:1411.4389 [cs]*, May 2016. arXiv: 1411.4389.

[6] J. A. Gregory. Shape Preserving Spline Interpolation. June 1985.

[7] Mitch Hill. THE UNCERTAINTY PRINCIPLE FOR FOURIER TRANSFORMS ON THE REAL LINE. page 17.

[8] Maryam Imani and Hassan Ghassemian. Curve fitting, filter bank and wavelet feature fusion for classification of PCG signals. In *2016 24th Iranian Conference on Electrical Engineering (ICEE)*, pages 203–208, May 2016. ISSN: null.

[9] Edward Kim and Kathleen F. McCoy. Multimodal Deep Learning using Images and Text for Information Graphic Classification. In *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '18, pages 143–148, Galway, Ireland, October 2018. Association for Computing Machinery.

[10] Kuan Liu, Yanen Li, Ning Xu, and Prem Natarajan. Learn to Combine Modalities in Multimodal Deep Learning. *arXiv:1805.11730 [cs, stat]*, May 2018. arXiv: 1805.11730.

[11] Alberto López, Francisco Javier Martin, David Yangüela, and Constantina Alvarez Peña. Development of a Computer Writing System Based on EOG. *Sensors*, 17:1505, June 2017.

[12] Raed Y. Mesleh, Harald Haas, Sinan Sinanovic, Chang Wook Ahn, and Sangboh Yun. Spatial Modulation. *IEEE Transactions on Vehicular Technology*, 57(4):2228–2241, July 2008. Conference Name: IEEE Transactions on Vehicular Technology.

[13] Michel Misiti, Yves Misiti, Georges Oppenheim, and Jean-Michel Poggi. *Wavelets and their Applications*. John Wiley & Sons, March 2013. Google-Books-ID: Ee-MYyvA5PDoC.

[14] Timothy J. O'Shea, Johnathan Corgan, and T. Charles Clancy. Convolutional Radio Modulation Recognition Networks. *arXiv:1602.04105 [cs]*, June 2016. arXiv: 1602.04105.

[15] G. Strang and T. Nguyen. *Wavelets and Filter Banks*. Wellesley-Cambridge Press, 1996.

[16] Jingyuan Wang, Ze Wang, Jianfeng Li, and Junjie Wu. Multilevel Wavelet Decomposition Network for Interpretable Time Series Analysis. *arXiv:1806.08946 [cs, eess, stat]*, June 2018. arXiv: 1806.08946.

[17] Tong Wang and Qihang Lin. Hybrid Predictive Model: When an Interpretable Model Collaborates with a Black-box Model. *arXiv:1905.04241 [cs, stat]*, May 2019. arXiv: 1905.04241.

[18] Zhiguang Wang and Tim Oates. Encoding Time Series as Images for Visual Inspection and Classification Using Tiled Convolutional Neural Networks. page 7.

[19] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, 9(4):611–629, August 2018. Number: 4 Publisher: SpringerOpen.