Indian Institute of Technology Ropar

CP-301: Departmental Engineering Project

Deep Learning-Based Spectrum Sensing for Dynamic Spectrum Access

Shashwat Srivastava 2021EEB1210

Siddhartha Arora 2021EEB1213

Submitted to

Dr. Satyam Agarwal

May 10, 2024

Abstract

Efficient spectrum sensing is crucial for cognitive radios to manage available frequency bands effectively. In this study, we propose a new method for spectrum sensing using deep learning, specifically convolutional neural networks (CNNs). We train our model on a diverse dataset containing different signal types and noise data, enabling it to adapt to new signals. We also explore transfer learning techniques, like using pre-trained models, to improve detection accuracy further. Our experiments confirm that our approach outperforms traditional methods, showing better accuracy in detecting real-world signals. By automating feature extraction and learning from data, our model simplifies the process and reduces the need for specialized knowledge. Overall, our research contributes to advancing spectrum sensing methods, making them more efficient and accessible for cognitive radio applications.

Keywords

Spectrum Sensing, Spectrogram, Deep Learning, Convolutional Neural Network, Transfer Learning

Acknowledgement

We extend our heartfelt gratitude to Dr. Satyam Agarwal, our project supervisor, for his invaluable guidance, encouragement, and support throughout the development of our project on Deep Learning-Based Spectrum Sensing for Dynamic Spectrum Access. His expertise and insights have been instrumental in shaping our ideas and approach. Our appreciation also goes to the faculty and staff of the Electrical Engineering Department for their support and provision of resources. Their dedication to education and research has equipped us with the necessary tools and knowledge to undertake this project effectively. We are grateful to our peers for their feedback, support, and encouragement throughout this endeavor. Their contributions have helped refine our ideas and maintain motivation. Lastly, we express our heartfelt thanks to our families and friends for their unwavering support, love, and encouragement. Their belief in our abilities has been a constant source of motivation, and we are grateful for their presence throughout this journey. We extend our sincere gratitude to all those who supported us, and we hope that our work will contribute to the advancement of spectrum sensing technology and its responsible utilization in society.

Motivation

As the demand for wireless spectrum continues to soar, traditional spectrum allocation approaches are proving insufficient to meet evolving needs. Conventional spectrum sensing methods, while effective to a degree, often struggle with issues like noise sensitivity and computational complexity. In contrast, deep learning techniques, particularly CNNs applied to spectrogram analysis, offer a new frontier in spectrum sensing. By leveraging the capabilities of deep learning, we aim to develop innovative solutions that enhance the efficiency and adaptability of spectrum sensing systems. Our project seeks to address these challenges head-on, driving forward the development of more resilient and intelligent wireless communication technologies.

1 Introduction

Wireless spectrum, encompassing radio waves within the electromagnetic spectrum, is fundamental for wireless communication, facilitating data transmission vital for various applications, including IoT systems. With the escalating demand for wireless connectivity, particularly fueled by the proliferation of IoT devices, efficient spectrum management becomes paramount. Spectrum sensing, involving the periodic monitoring of specific frequency bands to detect primary user activity, is essential for effective spectrum utilization.

Advancements like 5G technology aim to expand available spectrum and optimize its utilization to meet evolving wireless needs. Cognitive radio networks (CRNs) represent a pioneering technology designed to enhance spectrum efficiency. CRNs grant secondary users the privilege to access licensed spectrum when unused by primary users, under the condition of vacating it upon primary user demand. Spectrum sensing emerges as a critical capability for cognitive radios, enabling them to accurately assess the status of target spectrum.

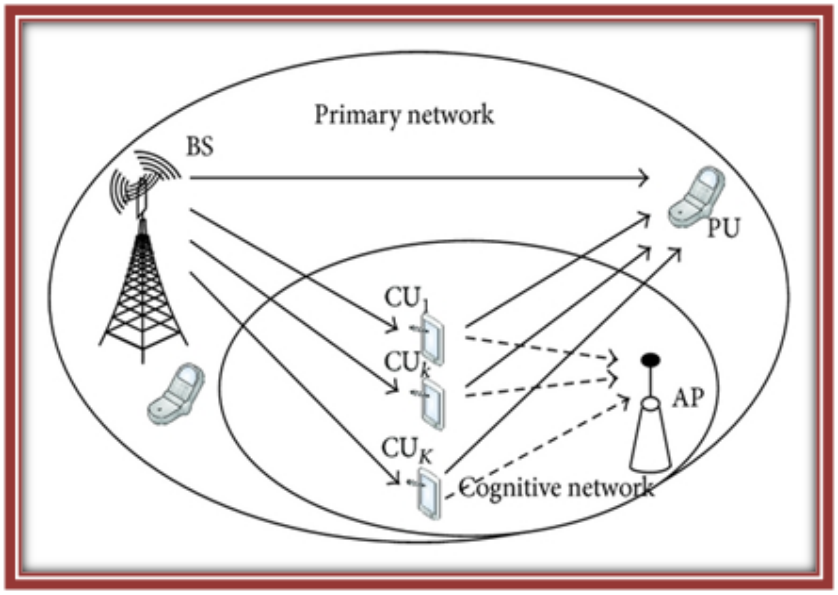

Figure 1: Cognitive Radio Network Architecture

However, spectrum resources are scarce, necessitating efficient spectrum utilization to address this challenge. Extensive research has been devoted to spectrum sensing, resulting in the development of numerous algorithms. While traditional methods rely on complex decision statistics and domain-specific knowledge, leveraging machine learning and deep learning offers a promising alternative. With the advances in deep learning and neural networks, it is now possible to create simple yet powerful networks able to perform classification, forecasting and object detection tasks that were previously highly complex and sometimes even infeasible. In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural

networks, most commonly applied to analyzing visual imagery. Deep learning techniques, particularly convolutional neural networks (CNNs), demonstrate remarkable potential in various tasks, including image analysis and, increasingly, non-image problems like signal processing. For instance, a signal can be converted to a spectrogram, which is a chart that shows the amount of each frequency at each time in an audio file. It has been shown that this approach can beat state-of-the-art environmental signal detection models.

Spectrograms have gained popularity due to their compatibility with standard 2D convolutional neural networks (CNNs) like UNet, offering a straightforward approach for analysis. This trend is evident in research papers that treat "frequency bins" as spatial dimensions rather than channels, essentially treating the spectrogram as a grayscale image. However, this method assumes frequency shift invariance, which is not entirely accurate, as only the time dimension demonstrates shift invariance. For instance, altering the frequencies of a piano sound by 50 Hz doesn't result in a higher-pitched piano sound but rather produces an unfamiliar, robotic tone. Despite these nuances, many researchers favor using standard image CNNs due to their simplicity and effectiveness. This trend underscores the widespread adoption of spectrogram-based CNNs, primarily driven by their ease of implementation and promising results, making them an appealing choice for researchers in the field.

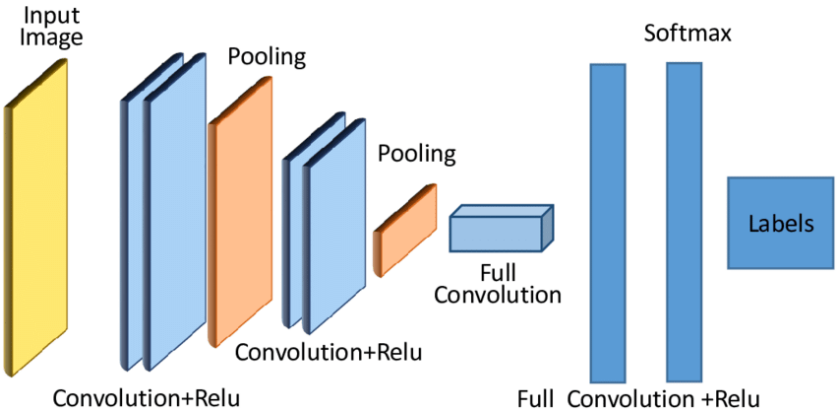

Figure 2: Generic CNN Architecture

In a stacked configuration, each layer of convolutional neural networks (CNNs) progressively enhances the understanding of individual features by expanding their contextual scope. This expansion is quantified by the "receptive field," which measures the area of input data each feature perceives, considering factors such as layer count, kernel size, and stride. Consequently, while initial layers focus on local changes, deeper layers develop a broader perspective, capturing long-distance dependencies within the data.

In the context of image analysis, CNNs exhibit parallels to generalized wavelet analysis. When applied to one-dimensional data, such as in the Fast Fourier Transform (FFT), CNNs function analogously to chirplet analysis. This analogy highlights CNNs' capacity to discern patterns in

frequency component variations over time, enabling the extraction of high-level features crucial for tasks like signal processing and classification. To address these challenges, we aim to design a CNN architecture optimized for both accuracy and computational efficiency, advancing the field of spectrum sensing and cognitive radio technology.

2 Traditional Works

Various techniques are employed in traditional spectrum sensing, each with its distinct advantages and challenges. Energy detection [1], renowned for its simplicity, has enjoyed widespread application but is susceptible to uncertainties stemming from noise. Cyclostationary feature-based detection [2], on the other hand, excels in low signal-to-noise scenarios but demands substantial computational resources and prior signal information. Eigenvalue-based detection [3], relying on eigenvalue calculations, exhibits robustness against noise power uncertainties. Meanwhile, the frequency domain entropy-based approach [4] leverages disparities in frequency domain distributions between signal and noise. These methodologies collectively offer a spectrum of options for spectrum sensing, each catering to different operational requirements and environmental conditions.

In the field of spectrum sensing, Machine Learning (ML) has gained prominence, especially in cooperative sensing. For example, in [5], researchers proposed various cooperative sensing algorithms using support vector machine (SVM), weighted K-nearest neighbor, K-means clustering, and Gaussian mixture model. They measured signal energy as a feature. Similarly, [6] introduced a collaborative sensing method using Convolutional Neural Networks (CNNs) to improve performance and reduce complexity. [7] provided an overview of kernel-based learning (KBL) methods in cognitive radio networks, exploiting their signal processing capabilities. Additionally, [8] and [9] developed sensing techniques using Artificial Neural Networks (ANNs), considering features like energy and likelihood ratio test statistics. [10] explored CNN architectures for spectrum sensing, focusing on cyclostationary features. These studies typically extract features first and then classify them using neural networks. Also, in [11], researchers extended the use of deep neural networks, such as CNNs and Recurrent Neural Networks (RNNs), to detect radar signals in the 3.5 GHz band based on spectrogram analysis.

This report addresses the broader spectrum sensing issue, rather than focusing on detecting specific signals. We frame the problem as a classification task with two categories and employ deep learning to solve it. During training, we expose the network to diverse signal types and noise data, aiming to equip the model to handle various unknown signals. As is common in traditional spectrum sensing, we assume the presence of additive white Gaussian noise (AWGN). Our proposed method's effectiveness will be thoroughly assessed through extensive experimental evaluation on various kinds of data.

3 Problem Formulation

We introduce spectrum sensing as a binary classification task and suggest a deep learning-based approach for spectrum sensing. Considering a noisy signal $y(t)$, is there an underlying signal $x(t)$ present within $y(t)$? Put differently, can $y(t)$ be represented as $x(t) + n(t)$, where $n(t)$ denotes noise? This question can be resolved through the correlation between $y(t)$ and $x(t)$. If the correlation exhibits significant magnitude at a specific time delay τ, we can confidently assert an affirmative answer.

It is noteworthy that when dealing with symmetric signals, convolution and cross-correlation operations coincide; this scenario is prevalent in certain domains of Digital Signal Processing (DSP). Our method leverages the spectrogram of the signal as CNN input and incorporates diverse signal and noise data for network training. By harnessing deep learning, the method autonomously learns signal and noise features from the data, eliminating the need for manual feature extraction. Consequently, it exhibits the capability to detect even untrained signals effectively.

We conduct extensive experiments to verify the performance of the proposed method. More specifically, we follow a systematic approach derived from various considerations:

- **Filter Selection:** Filters with odd sizes are chosen to ensure a central point for convolutional operations.
- **Channel Consideration:** We acknowledge the trade-off between the number of channels and the risk of overfitting. Hence, we increase the number of channels cautiously, as it directly impacts feature learning and the propensity for overfitting.
- **Pooling Strategy:** Max pooling is primarily utilized for downsampling, maintaining spatial information while reducing dimensionality.
- **Normalization and Activation:** Batch normalization is applied before dropout and after the activation function, adhering to best practices for network stability and learning dynamics.
- **Filter Evolution:** We initiate with small filters to capture localized information effectively, gradually increasing filter width to encompass broader features.
- **Channel Management:** Initially, we keep the number of channels low, gradually scaling them up as the network evolves.
- **Layer Expansion and Overfitting Control:** We iteratively add layers until signs of overfitting become apparent. Regularization techniques are then introduced to mitigate this issue effectively.

- **Model Architecture Inspiration:** Drawing inspiration from classic models, we explore trends in layer configurations such as Conv-Pool-Conv-Pool or Conv-Conv-Pool-Conv-Conv-Pool, as well as variations in channel progression (e.g., 32–64–128) and filter sizes.

4 Technologies used

1. *Python:* Python serves as the backbone for analysis due to its simplicity, extensive libraries, and community support. With libraries like NumPy, Pandas, and Matplotlib, it provides powerful tools for data manipulation, visualization, and preprocessing, essential for this project.

2. *MATLAB:* MATLAB was used to generate training data. MATLAB offers a range of pre-written utilities, such as the "fskmod" function for frequency-shift keying modulation and the "spectrogram" function for generating spectrogram images. In MATLAB, the spectrogram function is part of the Signal Processing Toolbox.

3. *TensorFlow:* TensorFlow, an open-source machine learning framework developed by Google, provides robust support for building and training deep neural networks. With TensorFlow's high-level APIs like Keras, it's easier to construct complex architechture. Other substitute could have been Pytorch. But we chose Tensorflow because TensorFlow had a better community support.

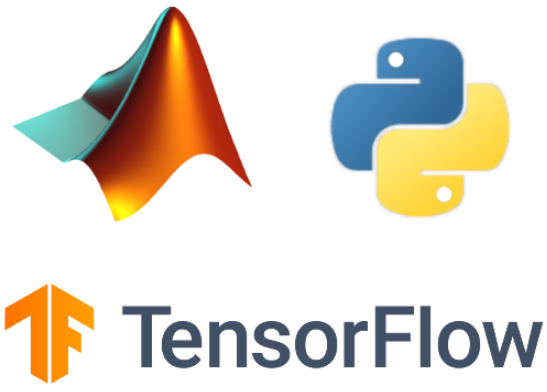

Figure 3: MATLAB, Python & Tensorflow

5 ConvNets based Solution

5.1 Background

Convolutional Neural Networks (CNNs) represent a specialized class of neural networks tailored for processing data structured in grid-like arrangements, predominantly utilized for image analysis tasks. The nomenclature "convolutional" reflects the utilization of convolution, a specific form of linear operation, within the network architecture. CNNs stand out as neural networks employing convolutional operations in at least one layer, replacing the more general matrix multiplications common in traditional architectures.
Convolution brings forth three pivotal concepts that enhance machine learning systems: sparse interactions, parameter sharing, and equivariant representations. These notions contribute to the efficiency and effectiveness of CNNs. Notably, CNNs require fewer parameters compared to their dense-layer counterparts, thus offering computational advantages. Furthermore, convolution facilitates handling inputs of varying dimensions, providing adaptability crucial for diverse data types and applications. Since image data is structured i.e. it has edges and shapes and further there is translation and scale invariance. CNN emulates human vision system. Each component of a CNN tries to learn to extract different features. Two main components of a CNN are convolution and pooling. Hence, they perform better than multi-layer perceptron.

5.2 Proposed Model

We propose a convolutional neural network (CNN) model for spectrum sensing, designed to classify signals into binary categories. The model architecture, depicted below, is tailored to effectively capture signal features through successive convolutional and pooling layers, followed by dense layers for classification.

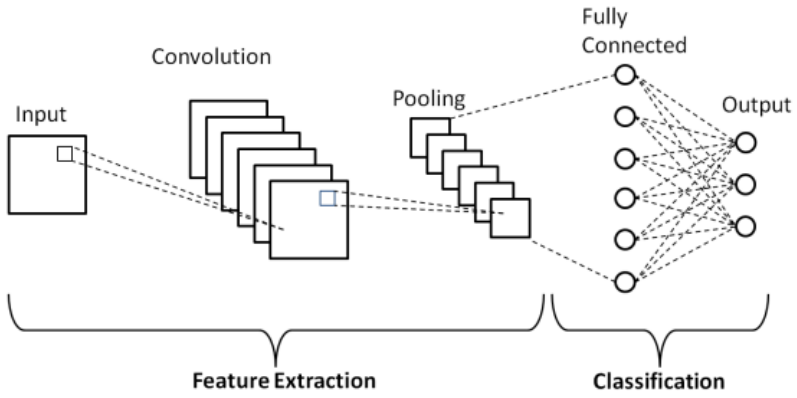

Figure 4: Binary Classification using CNNs

Table 1: Summary of Model Architecture

Layer Type	**Output Shape**	**Parameters**
Conv2D	(None, 875, 656, 32)	896
MaxPooling2D	(None, 437, 328, 32)	0
BatchNormalization	(None, 437, 328, 32)	128
Conv2D	(None, 437, 328, 64)	18496
MaxPooling2D	(None, 218, 164, 64)	0
BatchNormalization	(None, 218, 164, 64)	256
Conv2D	(None, 218, 164, 64)	36928
MaxPooling2D	(None, 109, 82, 64)	0
BatchNormalization	(None, 109, 82, 64)	256
Conv2D	(None, 109, 82, 128)	73856
MaxPooling2D	(None, 54, 41, 128)	0
BatchNormalization	(None, 54, 41, 128)	512
Conv2D	(None, 54, 41, 128)	147584
MaxPooling2D	(None, 27, 20, 128)	0
BatchNormalization	(None, 27, 20, 128)	512
Flatten	(None, 69120)	0
Dense	(None, 512)	35389952
Dropout	(None, 512)	0
Dense	(None, 1)	513
Total Trainable Parameters		**35,669,057**
Non-trainable Parameters		**832**

The model comprises five convolutional layers followed by max-pooling layers and batch normalization. These layers are designed to progressively extract hierarchical features from the input spectrogram of signals. Subsequently, the flattened output is passed through dense layers with dropout regularization to prevent overfitting, culminating in a single neuron with a sigmoid activation function for binary classification.

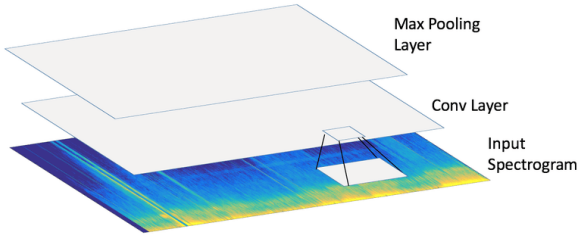

Figure 5: CNN with spectrogram as input

The model is trained using the Adam optimizer with a binary cross-entropy loss function. The dataset is split into training and validation sets, with a total of 20 epochs utilized for training. The training process involves iterating over the training dataset for multiple epochs, optimizing the model parameters to minimize the loss function while monitoring performance on the validation set to prevent overfitting.
Binary cross-entropy, also known as logistic loss, is commonly used as the loss function for binary classification problems. It is defined as:

$$\text{Logistic loss} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)] \tag{0.1}$$

where:

- N is the total number of samples,
- y_i is the true label of the i-th sample (0 or 1),
- p_i is the predicted probability that the i-th sample belongs to the positive class.

6 Code Explanation:

The code can be divided into several parts -

1. **Mounting Google Drive:** To access files stored in Google Drive, we needed to mount it to the Colab environment. This allowed us to read, write, and manipulate files seamlessly.

```
[ ]  # Importing the necessary module from the Google Colab library to mount Google Drive.
     from google.colab import drive

     # Mounting the Google Drive to the Colab environment.
     drive.mount('/content/drive')
```

Figure 6: Code for Mounting Drive to Colab

2. **Extracting Dataset Zip File:** After mounting, we extracted the dataset stored in a zip file located in our Google Drive. This dataset is essential for our project, and we later used it for training & validation.

```
[ ]  import os

     # Importing the 'zipfile' module to work with zip files.
     import zipfile

     # Specifying the path of the zip file in Google Drive and the directory to extract the contents.
     zip_file_path = '/content/drive/My Drive/dataset.zip'
     extraction_path = '/content/dataset'

     # Extracting the contents of the zip file to the specified directory.
     with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
         zip_ref.extractall(extraction_path)
```

Figure 7: Code for extracting dataset zip file

3. **Importing Necessary Libraries:** Here, we imported all the required libraries for our project. These libraries like pandas, matplotlib, seaborn, tensorflow, etc include tools for data manipulation, visualization, deep learning model creation, and more.

```
     # Importing 'numpy' library for numerical computations.
     import numpy as np

     # Importing 'pandas' library for data manipulation and analysis.
     import pandas as pd

     # Importing TensorFlow library for building and training deep learning models.
     import tensorflow as tf

     # Importing the 'ImageDataGenerator' class from Keras for data augmentation.
     from keras.preprocessing.image import ImageDataGenerator

     # Importing data visualization libraries.
     import seaborn as sns
     import plotly.express as px
     import matplotlib.pyplot as plt

     # Importing necessary modules from Keras for building the model.
     from keras.models import Sequential, load_model
     from keras.layers import Dense, GlobalAvgPool2D, Dropout
```

Figure 8: Code for importing libraries

4. **Image Data Generator:** Here, we define an ImageDataGenerator object named gen. This object is responsible for generating batches of image data with real-time data augmentation. The parameters specified include:
   - **rescale:** Rescales the pixel values of the images to the range [0, 1] by dividing each pixel value by 255.

- **validation_split:** Splits the data into training and validation sets, with 80% of the data used for training and 20% for validation.

We also define the rootpath here.

```
[ ]  # Image Data Generator for data preprocessing and augmentation.
     gen = ImageDataGenerator(
         rescale=1./255,           # Rescaling pixel values to the range [0, 1].
         validation_split=0.2      # Splitting the data into training and validation sets.
     )

[ ]  rootpath = '/content/dataset/dataset' # defining the rootpath
```

Figure 9: ImageDataGenerator Object

5. **Flow From Directory:** The flow_from_directory method generates batches of augmented data from image files in a directory. Here, we create training & validation datasets named train_ds and valid_ds using the following parameters:
   - **rootpath:** The directory path containing the image files.
   - **target_size:** Resizes the images to the specified dimensions of (875, 656) pixels.
   - **subset:** Specifies that this dataset is a subset of the data, in this case, the training set.
   - **batch_size:** The number of samples in each batch.
   - **shuffle:** Shuffles the data randomly.
   - **class_mode:** Defines the type of labels, which is set to 'binary' indicating binary classification.

```
[ ]  # Generating batches of augmented data from image files in the directory for training.
     train_ds = gen.flow_from_directory(
         rootpath,                    # Directory path containing the image files.
         target_size=(875, 656),      # Resizing images to (875, 656) pixels.
         subset='training',           # Specifying that this is the training subset.
         batch_size=32,               # Number of samples in each batch.
         shuffle=True,                # Shuffling the data randomly.
         class_mode='binary'          # Labels are binary (binary classification).
     )
```

Figure 10: Code for creating training-validation datasets

6. **Get Class Names:** To retrieve the class names from the dataset directory, we use the os.listdir function to list all the directories present in the root path and select the first two directories as class names. These class names represent the categories or labels of the images in the dataset.

7. **Model Architecture Definition:** The following code defines the architecture of the convolutional neural network (CNN) model using TensorFlow and Keras. Below are the key features of the model:

```
# Retrieve class names from the dataset directory
class_names = os.listdir(rootpath)[:2]
class_names
```

Figure 11: Gets us class names

- Utilizes odd-sized filters (3x3) for convolutional layers.
- Applies batch normalization before dropout and after activation to improve convergence and speed up training.
- Initiates with smaller filter sizes and gradually increases filter width to capture more complex features.
- Reduces the number of channels initially and gradually increases them in subsequent layers.
- Adds padding='same' parameter to convolutional layers to preserve border information.
- Implements multiple convolutional layers followed by max-pooling layers to downsample the feature maps.
- Incorporates a flattening layer to convert the 2D feature maps into a 1D vector.
- Includes dropout regularization with a dropout rate of 0.4 to prevent overfitting.
- Concludes with a dense layer with a sigmoid activation function to output binary classification results.

```
from tensorflow import keras
from tensorflow.keras import layers

# Define the model architecture
model = keras.Sequential([
    layers.Conv2D(32, kernel_size=(3, 3), activation="relu", input_shape=(875, 656, 3), padding='same'),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.BatchNormalization(),
    layers.Conv2D(64, kernel_size=(3, 3), activation="relu", padding='same'),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.BatchNormalization(),
    layers.Conv2D(64, kernel_size=(3, 3), activation="relu", padding='same'),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.BatchNormalization(),
    layers.Conv2D(128, kernel_size=(3, 3), activation="relu", padding='same'),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.BatchNormalization(),
    layers.Conv2D(128, kernel_size=(3, 3), activation="relu", padding='same'),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.BatchNormalization(),
    layers.Flatten(),
    layers.Dense(512),
    layers.Dropout(0.4),
    layers.Dense(1, activation="sigmoid"),
])

# Display the model summary
model.summary()
```

Figure 12: Code for Model Architecture

8. **Model Training:** The code below compiles and trains the defined CNN model using the specified parameters. Here's a breakdown of the process:

- **Epochs:** The number of epochs is set to 12, indicating the number of passes through the entire training dataset during training.
- **Loss Function:** Utilizes binary cross-entropy loss, suitable for binary classification tasks.
- **Optimizer:** Employs the Adam optimizer, a popular choice for training neural networks due to its adaptive learning rate and momentum.
- **Metrics:** Evaluates model performance during training using accuracy as the evaluation metric.
- **Training:** The model is trained using the fit() function, providing the training and validation datasets (train_ds and valid_ds), along with the number of epochs.

```
epochs = 12

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=["accuracy"])

# Train the model
hist = model.fit(train_ds, validation_data=valid_ds, epochs=epochs)
```

Figure 13: Code for Model Training

9. **Training and Validation Visualization:** The code below plots the training and validation accuracy as well as the training and validation loss over the epochs. These plots provide insights into the model's performance and its training dynamics.

```
import matplotlib.pyplot as plt

# Get training and validation accuracy
train_accuracy = hist.history['accuracy']
val_accuracy = hist.history['val_accuracy']

# Get training and validation loss
train_loss = hist.history['loss']
val_loss = hist.history['val_loss']

# Plot training and validation accuracy
plt.plot(range(1, epochs+1), train_accuracy, label='Training Accuracy')
plt.plot(range(1, epochs+1), val_accuracy, label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Plot training and validation loss
plt.plot(range(1, epochs+1), train_loss, label='Training Loss')
plt.plot(range(1, epochs+1), val_loss, label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Figure 14: Code for accuracy & loss visualisation

7 Performance Evaluation

7.1 Generating Simulated Data

In the domain of spectrum sensing, obtaining high-quality labeled data is of utmost importance, as it forms the foundation for training and evaluating machine learning models effectively. This report also explores this critical aspect in depth, presenting our innovative methodology for creating synthetic data. Through careful construction of scenarios that encompass diverse signal and noise variations, we meet the complex requirements of spectrum sensing tasks. Our objective is to generate an adaptable dataset that support robust model development and rigorous performance assessment in the dynamic landscape of spectrum sensing. We had a 60-40 ratio in MATLAB code so that "imbalanced" case doesn't arise.

7.2 Data Generation Code Explanation

The flow of synthetic spectrogram data generation is explained as follows:

1. **Setting Parameters:** First, we define the parameters required for signal generation. Speficially, the following parameters:
   - **modulation_orders:** Specifies the modulation order. (2, 4, 8, 16...).
   - **nsamp:** Samples per symbol.
   - **Fs:** Samples rate in Hertz.
   - **num_symbols:** Number of symbols.
   - **num_samples:** Number of samples to generate.

```
% Set parameters
modulation_orders = [2, 4, 8]; % Modulation orders to randomly select from
nsamp = 8;          % Samples per symbol
Fs = 200e3;         % Sample rate (Hz)
num_symbols = 1000; % Number of symbols
num_samples = 3000;   % Number of samples to generate

% Create directories for the dataset
mkdir dataset
mkdir dataset/signal
mkdir dataset/noise
```

Figure 15: Code for parameters and directories

2. **Creating Directories:** Two directories, namely "signal" and "noise" are created to store the spectrogram images for signal+noise and noise respectively.

3. **Generating Data Samples:** A random integer is chosen between 0 and 1. if the random integer is $\geq$0.4, signal is generated, otherwise only noise is generated. In case of signal,

random binary stream is generated, SNR is chosen at random in the range [-10, 20] (dB), Frequency separation is also chosen at random in the range [12.5, 25] (KHz).

```
% Loop to generate data samples
for i = 1:num_samples
    isSignal = rand() > 0.4; % Randomly decide whether it's a signal or noise
    if isSignal
        % Randomly select modulation order
        M = modulation_orders(randi([1, length(modulation_orders)]));

        % Generate random binary symbols (0 or 1)
        freqsep = genRandom(12.5e3, 25e3, 1); % Ensure freqsep <= Fs / (2 * M)
        SNR_dB = genRandom(-10, 20, 1);
        binary_symbols = randi([0 M-1], num_symbols, 1);
```

Figure 16: Code for Signal Generation

4. **Applying FSK and adding AWGN:** FSK Modulation is applied using the fskmod function and Additive White Gaussian Noise (AWGN) is added using the awgn function subsequently.

```
% Apply FSK modulation
fsk_modulated_signal = fskmod(binary_symbols, M, freqsep, nsamp, Fs);

% Add white Gaussian noise
fsk_modulated_signal_with_noise = awgn(fsk_modulated_signal, SNR_dB, 'measured');
```

Figure 17: Applying FSK and adding AWGN

5. **Only Noise case:** In the else case, when the random integer is less than 0.4, only noise is generated. This is done by using the randn function. Finally, the spectrogram is plotted and the image saved in the respective directory. This whole procedure happens over a loop of num_samples.

```
else
    % Generate spectrogram for only noise
    noise_only = randn(num_symbols * nsamp, 1); % Generate white Gaussian noise

    % Plot the spectrogram
    figure;
    spectrogram(noise_only, hann(256), 128, 256, 2*Fs, 'yaxis');
    set(gca, 'Visible', 'off');
    colorbar('off');

    % Save the spectrogram with appropriate labels
    filename = sprintf('dataset/noise/noise_%d.png', i);
    saveas(gcf, filename);
    close(gcf); % Close the figure to avoid cluttering
```

Figure 18: Code for generating only noise

We have two types of datasets broadly-

- *Fixed Modulation Order:* Our signal generation methodology encompasses the synthesis of diverse modulation schemes and channel conditions to create a robust dataset for

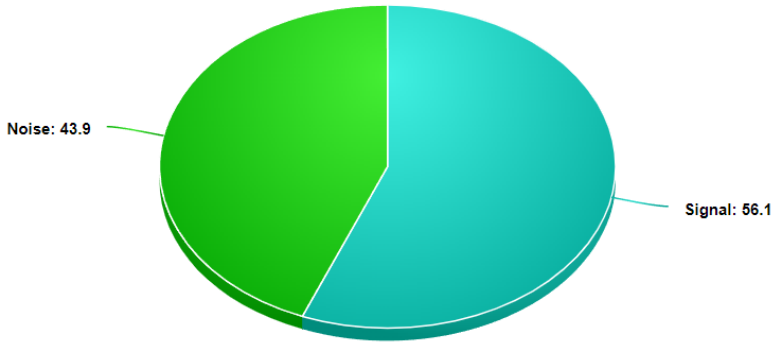

Figure 19: Class Distribution

spectrum sensing. We begin by defining crucial parameters such as modulation order, samples per symbol, and sample rate. Each signal sample is generated by randomly selecting modulation parameters, including frequency separation and signal-to-noise ratio. The generated binary symbols are then modulated using frequency-shift keying (FSK), followed by the addition of white Gaussian noise to simulate realistic channel conditions.

- *Varying Modulation Order:* Expanding upon the initial signal generation approach, we introduce variability in modulation orders and signal-to-noise ratios to create a more diverse dataset. By randomly selecting modulation orders from a predefined set and adjusting parameters accordingly, we simulate a broader range of signal scenarios. This extension allows for the testing of spectrum sensing algorithms under various modulation schemes and noise levels, ensuring their robustness and generalization capabilities. Enhanced spectrogram visualization techniques accommodate the diverse signal characteristics, providing a comprehensive dataset for training and evaluation purposes.

7.3 Training Process

Training accuracy is a crucial metric that reflects the performance of machine learning models during the training process. It measures the proportion of correctly classified samples relative to the total number of training samples. Here, we present a detailed analysis of the training accuracy achieved during the training of our spectrum sensing model over 20 epochs.
To visually represent the training accuracy and its comparison with validation accuracy for fixed modulation order of 4 over epochs, we provide graphs depicting the evolution of these metrics throughout the training process. The x-axis represents the number of epochs, while the y-axis indicates the accuracy values. These graphs serve as valuable tools for assessing the model's learning behavior, identifying overfitting or underfitting tendencies, and determining the optimal number of epochs for training.

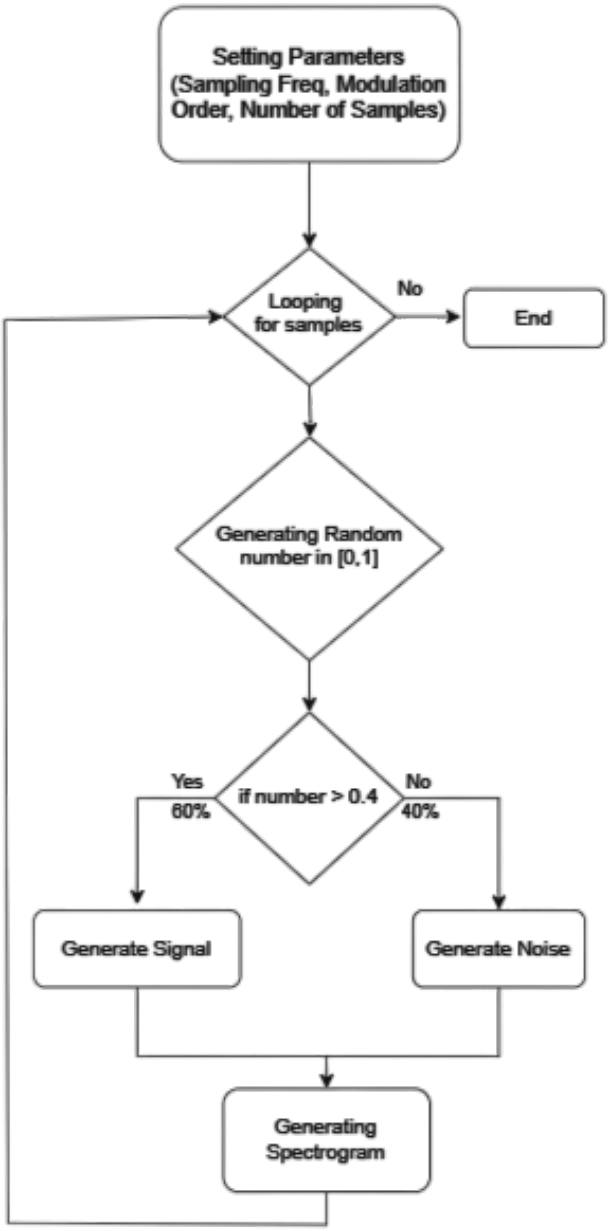

Figure 20: Program flow for synthetic data generation

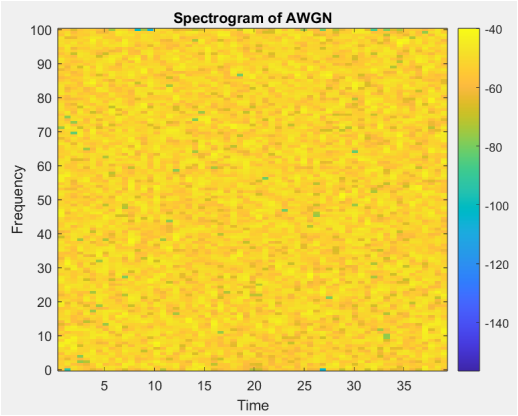

Figure 21: Synthetic Noise Spectrogram (AWGN)

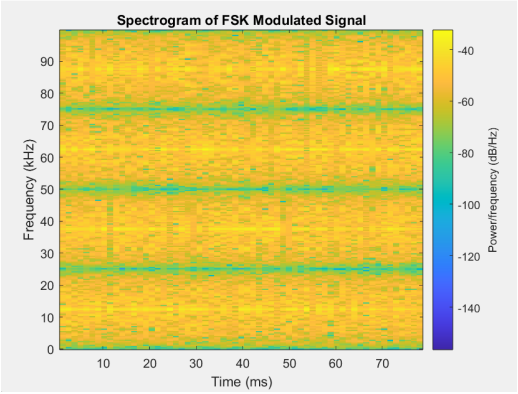

Figure 22: Signal Spectrogram

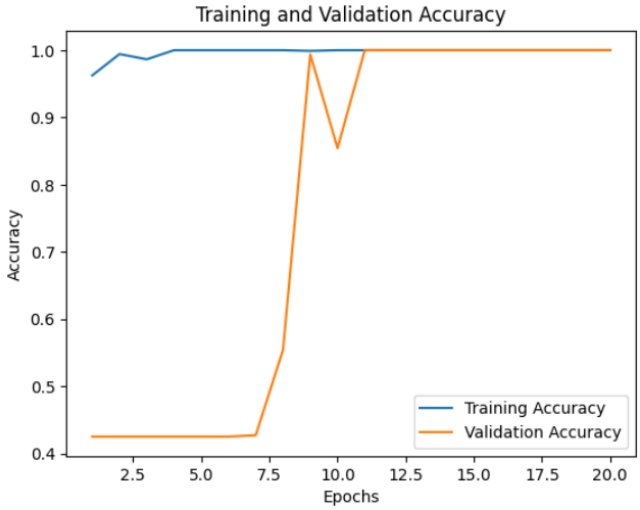

Figure 23: Training and validation accuracy for fixed modulation order

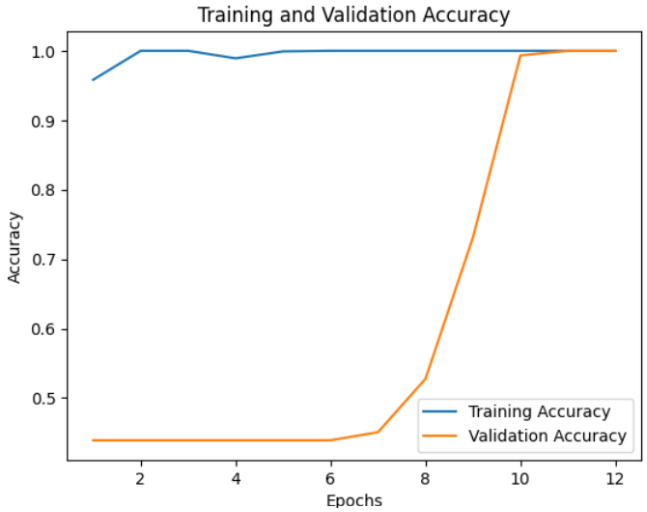

Figure 24: Training and validation accuracy for varying modulation order

We present two graphs here in Fig 23 & 24 which show the training vs validation accuracy for both types of data (fixed and varying modulation order). The training and validation loss decreases considerably as well after a certain number of epochs for both types of datasets.

7.4 Transfer Learning for better results

Incorporating pre-trained models like the Inception module can provide significant advantages in terms of feature extraction and model performance. At the core of GoogleNet lies the Inception module, designed to tackle the challenge of capturing features across different scales while managing computational complexity. This module uses multiple convolutional filters of different sizes (like 1x1, 3x3, and 5x5) within the same layer. By doing so, it can capture fine details with smaller filters and broader patterns with larger ones. Additionally, 1x1 convolutions are used to reduce dimensionality, which lowers the number of input channels and, consequently, the computational burden.

The outputs of these diverse filters are then combined along the depth dimension by concatenation. This parallel approach ensures that the network can learn features at both local and global levels, resulting in more robust representation capability.

We leverage the InceptionV3 model pre-trained on ImageNet and perform transfer learning by adding a few additional layers for task-specific adaptation. The base InceptionV3 model is loaded without the fully connected layers (include_top=False), and its weights are frozen to prevent re-training of the pre-trained layers. We then construct a sequential model consisting of the InceptionV3 base model followed by a global average pooling layer, dense layers with ReLU activation, dropout regularization, and a final dense layer with sigmoid activation for binary classification.

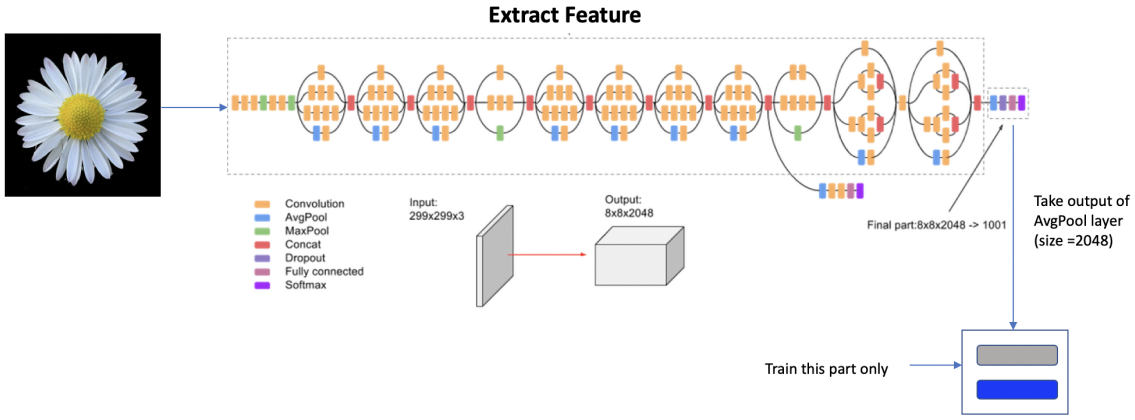

Figure 25: Inception V3 Model Architecture

The model is compiled with binary cross-entropy loss and the Adam optimizer, and accuracy is used as the evaluation metric. Early stopping and model checkpointing callbacks are employed to monitor the validation loss and ensure model convergence while preventing overfitting. The model is trained on the provided dataset for 20 epochs, with validation data used for evaluation during training.

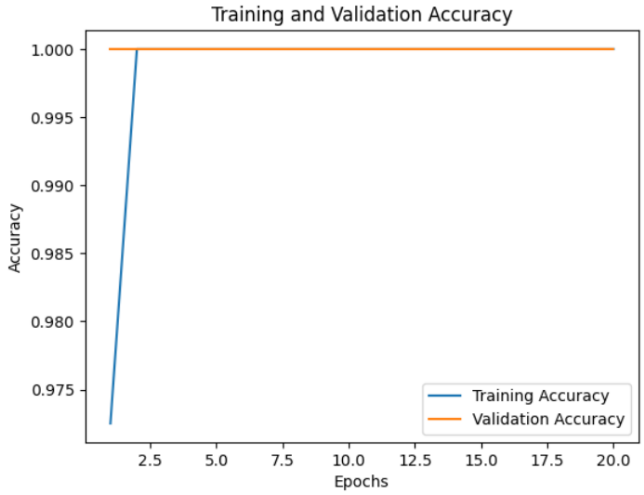

Figure 26: Training and validation accuracy using Inception module

While the Inception module demonstrates impressive performance on the provided dataset, it's essential to consider the computational resources required for training and inference. Custom models, especially simpler architectures, often require fewer computational resources, making them more suitable for deployment on resource-constrained devices or in real-time applications. While the Inception module showcases remarkable performance in terms of convergence speed and accuracy, the custom model offers unique advantages in interpretability, flexibility, and resource efficiency. It shows quite good results as well.

8 Testing on Field data

8.1 Background

HackRF One, developed by Great Scott Gadgets, is a versatile software-defined radio (SDR) transceiver. Designed by Michael Ossmann, it gained popularity after a successful Kickstarter campaign in 2014. This open-source device allows users to both send and receive signals, making it appealing to hackers, amateur radio enthusiasts, and cybersecurity professionals.

Figure 27: HackRF One - SDR transceiver

Capable of operating across a broad frequency range from 1 MHz to 6 GHz, HackRF One offers a maximum output power of 15 dBm, depending on the frequency band. Equipped with features like an SMA antenna port, clock input/output SMA ports, and a USB 2.0 port, it seamlessly integrates with popular SDR software like GNU Radio and SDR.
HackRF One has gained traction as a preferred tool for security research, frequently appearing in discussions and presentations at major information security conferences such as BlackHat, DEF CON, and BSides.

8.2 Field data collection

Field data collected using the HackRF Software Defined Radio (SDR) on 12th to 14th January 2024 is utilized for testing the proposed model. Antenna size used for this data collection was 1m. The data consists of In-phase and Quadrature (I-Q) samples captured at a sample rate of 20 MHz with a center frequency of 40 MHz. This dataset represents real-world radio frequency signals with inherent natural noise and interference.

Figure 28: Collecting real field data

8.3 Field data Results

The model trained on synthetic data is applied to this field dataset to evaluate its performance in a practical setting. Remarkably, the model demonstrates robustness and accuracy in classifying signals even in the presence of real-world complexities. Through rigorous testing and validation, the model consistently provides correct results, showcasing its effectiveness in real-world scenarios. Here is what we get:

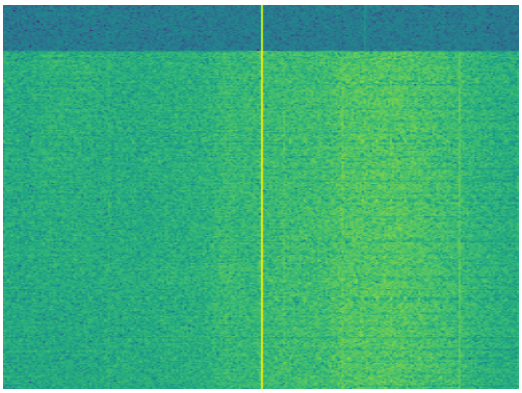

Figure 29: Spectrogram of the field data collected on 12-14 January 2024.

The successful application of the model on field data underscores its potential for various applications in wireless communication, spectrum analysis, and signal processing. Moreover, it highlights the reliability and adaptability of the proposed approach in handling diverse datasets and challenging environments.

After putting our trained model through thorough testing with real-world data, we've achieved top-notch performance results. This hands-on validation really drives home the effectiveness and practicality of our model, showing it's ready to be used in spectrum sensing scenarios.

```python
import cv2
import numpy as np

# Load and preprocess the spectrogram image
image_path = "/content/drive/MyDrive/spectrogram.png"
image = cv2.imread(image_path)
image = cv2.resize(image, (656, 875))
image = image / 255.0
image = np.expand_dims(image, axis=0)

# Make predictions using the trained model
predictions = model.predict(image)

# Get the predicted class
predicted_class = class_names[np.argmax(predictions)]

# Print the predicted class
print("Predicted class:", predicted_class)
```

```
1/1 [==============================] - 0s 28ms/step
Predicted class: signal
```

Figure 30: Results on field data

9 Future Ideas

Convolution is a mathematical operation central to signal processing. We explore the conventional convolution process, which involves sliding a kernel over an input volume to produce an output. However, this standard method can be computationally intensive due to the large number of multiplications required. To address this, we introduce depth-wise separable convolution, which splits the convolution process into two stages: depth-wise convolution and point-wise convolution.

In depth-wise convolution, we process each channel of the input separately. This saves a lot of time because we're not doing complex operations across all channels at once. Then, in point-wise convolution, we combine these results to get our final output. This split approach makes the whole process much faster and more efficient, which is great for tasks like deep spectrum sensing where speed matters.

10 Conclusion

In conclusion, our work presents a novel approach to spectrum sensing using deep learning techniques, particularly convolutional neural networks (CNNs). We address the challenge of

efficient spectrum utilization by modeling spectrum sensing as a binary classification task and leveraging the power of deep learning for signal detection.

By training our model on a diverse dataset comprising various signal types and noise data, we enable it to adapt to new and unseen signals effectively. Through rigorous experimentation, we validate the efficacy of our approach and demonstrate its adaptability to real-world signal detection scenarios. Moreover, our method showcases the effectiveness of transfer learning techniques in further enhancing detection performance, particularly when incorporating pre-trained models like the Inception module.

Our proposed CNN-based solution offers several advantages over traditional spectrum sensing methods. By automating feature extraction and learning directly from data, our model eliminates the need for manual feature engineering, thus reducing reliance on domain-specific knowledge. Additionally, our approach exhibits robustness to noise uncertainty and can adapt to non-ideal noise conditions, enhancing its performance in practical scenarios.

Furthermore, we provide insights into the design considerations and training processes involved in developing our spectrum sensing model. By carefully selecting model architecture, training parameters, and evaluation metrics, we ensure the effectiveness and reliability of our approach. Overall, our work contributes to advancing the field of spectrum sensing by harnessing the capabilities of deep learning and CNNs. By providing a robust and adaptable solution for spectrum sensing, we aim to facilitate more efficient spectrum utilization, thereby addressing the growing demand for wireless connectivity in various applications.

Bibliography

[1] F. F. Dighan, M.-S. Alouini, and M. K. Simon, "On the energy detection of unknown signals over fading channels," in Proc. IEEE ICC, 2003, vol.5, pp. 3575-3579.

[2] J. Lundén, S. A. Kassam, and V. Koivunen, "Robust nonparametric cyclic correlation-based spectrum sensing for cognitive radio," IEEE Transactions on Signal Processing, vol. 58, no. 1, pp. 38-52, 2010.

[3] Y. Zeng and Y.-C. Liang, "Eigenvalue-based spectrum sensing algorithms for cognitive radio," IEEE Transactions on Communications, vol. 57, no. 6, pp. 1784-1793, 2009.

[4] Y. Zhang, Q. Zhang, and S. Wu, "Entropy-based robust spectrum sensing in cognitive radio," IET Communications, vol. 4, no. 4, pp. 428-436, 2010

[5] K. M. Thilina, K. W. Choi, N. Saquib, and E. Hossain, "Machine learning techniques for cooperative spectrum sensing in cognitive radio networks," IEEE Journal on Selected Areas in Communications, vol. 31, pp. 2209–2221, November 2013.

[6] W. Lee, M. Kim, D. Cho, and R. Schober, "Deep sensing: Cooperative spectrum sensing based on convolutional neural networks," CoRR, vol. abs/1705.08164, 2017. [Online]. Available: http://arxiv.org/abs/1705.08164

[7] G. Ding, Q. Wu, Y.-D. Yao, J. Wang, and Y. Chen, "Kernel-based learning for statistical signal processing in cognitive radio networks: Theoretical foundations, example applications, and future directions," IEEE Signal Processing Magazine, vol. 30, no. 4, pp. 126-136, July 2013.

[8] M. R. Vyas, D. K. Patel, and M. Lopez-Benitez, "Artificial neural network based hybrid spectrum sensing scheme for cognitive radio," in 2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC), Oct 2017, pp. 1–7.

[9] Y. Tang, Q. Zhang, and W. Lin, "Artificial neural network based spectrum sensing method for cognitive radio," in 2010 6th International Conference on Wireless Communications Networking and Mobile Computing, 2010, pp. 1–4.

[10] D. Han, G. C. Sobabe, C. Zhang, X. Bai, Z. Wang, S. Liu, and B. Guo, "Spectrum sensing for cognitive radio based on convolution neural network," in 2017 10th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI), 2017, pp. 1–6.

[11] W. M. Lees, A. Wunderlich, P. Jeavons, P. D. Hale, and M. R. Souryal, "Deep learning classification of 3.5 GHz band spectrograms with applications to spectrum sensing," aiXiv: 1806.07745, 2018.