22.01.2024

EE435 Communications I

Term Project Phase 1 Report

Emir Atak 2515518

Alkım Bozkurt 2515675

1. Introduction

In this project, classification of analog modulated radio signals according to their modulation type is performed using a deep learning method, ResNet. The intended classification system will be able to distinguish 5 modulation types. To be acquinted with methodology behind deep learning approaches to accomplish signal classification, literature is reviewed starting from provided paper [1]. To use ResNet for classification, it must be trained by synthetic datasets. To generate synthetic signals which constitutes datasets, signal model parameters are determined according to [1] at first. Then, generation of datasets is performed for each modulation type under various signal to noise ratio (SNR) and carrier frequency offset (CFO) values. Then the classification performance of the ResNet model will be evaluated in different SNR and CFO scanarios. In this report, literature review about radio signal classification is given at first. Then, dataset generation is explained with determined signal model parameters.

2. Literature Review

Classification of radio signals has been very important working area for numerous radio sensing and communication systems. The classification has been accomplished by carefully hand-crafting specialized feature extractors for many years. Hand-crafting specialized feature extracting is a manual process in which some algorithms or mathematical formulations are tried to create to classify the raw data by extracting its specific features. To accomplish this, deep knowledge about the data, its properties and expertise are needed. To handle the classification, deep learning based methods are being used in recent years. Deep Learning (DL) techniques often use raw data directly. Residual Network (ResNet) is a deep learning architecture using a concept called residual learning to make it easier to train networks with many layers, avoiding the issues of vanishing gradients and degradation of accuracy [2]. Neural networks like ResNet automatically learn relevant features during training. To train the network, a robust dataset must be generated [3]. The success of ResNet depends directly on the quality of the dataset. DL methods increased the capacity and pace for feature learning compared to the traditional methods like hand-crafted feature extractors.

3. Data Generation

To train our DL model, a comprehensive dataset is crucial to ensure the model learns to distinguish various modulation types effectively. The synthetic dataset is typically generated by simulating signals for the given analog modulation types namely DSB-SC, DSB-WC, SSB-SC, SSB-WC and FM. We implemented the channel impairments which were mentioned in the previous chapter for a more reliable simulation of real-world data transmission.

$$s(t) = m(t)_I \cdot cos(2\pi f_c ct) - m(t)_Q \cdot sin(2\pi f_c t) \qquad [1]$$

or

$$s(t) = \{m(t)_I + jm(t)_Q\}\, e^{(-j2\pi f_c t)}$$

For a selected carrier frequency f_c , equation above is the time-domain modulated signal transmitted from the SDR. As it can be seen, the modulated signal *s(t)* is a complex signal which consists of in-

phase and quadrature components. For the given modulation types which we have generated signal, we constructed the message signal in the form of $m(t) = m_I(t) + jm_Q(t)$ to be able to utilize I/Q modulation. In other words, different modulation methods yield different in-phase and quadrature components. By arranging our in-phase and quadrature components with respect to configuration of each modulation type, we generated waveforms of that type of modulation.

We chose three different sound files to create our messages. To ensure a variety of frequency components, we selected music files rather than single-tone sinusoids. Furthermore, to increase the randomness of the dataset, we picked a random 4096 sample within these audio files as our message signal m(t).

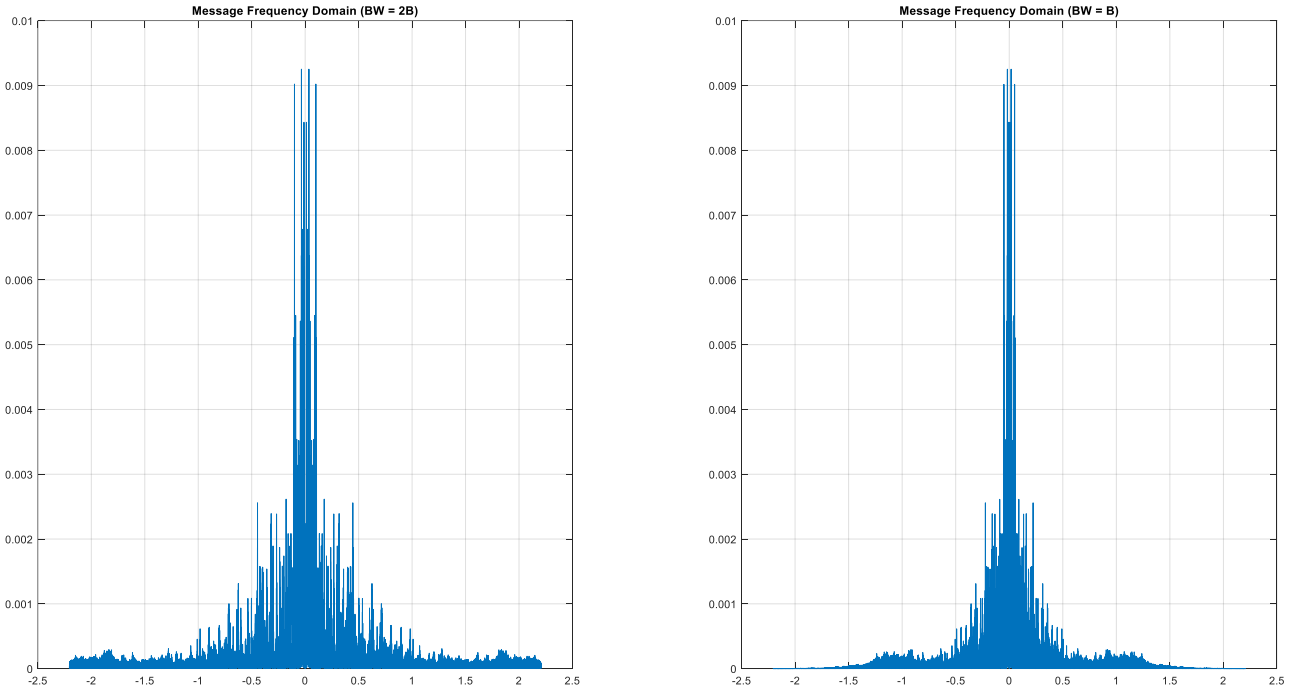


Figure 1 – Message FFT

To ensure the twice-bandwidth constraint, we up-sampled our audio signal with a sampling rate of twice the original sampling rate and we obtained a wider bandwidth of 2B.

In real life, there is always a probability that there is a frequency deviation between the frequencies of carriers which are used to modulate and demodulate the message signal. This frequency deviation is defined as carrier frequency offset. We modelled the difference between the frequencies of carriers for modulation and demodulation Δf as gaussian process whose mean is 0 and variance is 0.01 [1]. Because of the frequency deviation, message is not recovered perfectly.

$$u(t) = 2 * cos(2\pi(f_c + \Delta f)t + \emptyset) \qquad [2]$$

Equation 2 represents the clock signal for demodulation. As it can be seen from the equation it has CFO and phase-offset. If a demodulation is conducted by using signal *u(t)* and if a low-pass filter is applied to eliminate high order frequencies (~4*f_c) the recovered signal becomes

$$r(t) = \{m(t)_I + jm(t)_Q\}\, e^{(j2\pi\Delta f t)} \qquad [3]$$

If the one compare equation 1 and equation 3, it is obvious that the demodulated signal is the frequency shifted version of the original message signal. By applying an external distortion to our messages, we can simulate the effects of CFO.

To consider the thermal noise due to physical sensitivity of the devices, we simply implemented a AWGN process to our modulated signals. We have simulated several SNR scenarios from starting -20dB to 40 dB with 10dB steps, and we also simulated our channel under perfect noise (infinite SNR) conditions. In the below equation, r(t) is the received signal if we ignore path loss, and n(t) is AWGN process.

$$r(t) = s(t) + n(t) \qquad [4]$$

Under the **channel impairments** and other given requirements, we created our dataset with respect the modulation types from the previously described message namely *m(t)*.

Table 1: Modulated signal s(t).

Modulation Types	In-phase component $m_I(t)$	Quadrature component $m_Q(t)$	Modulated signal $s(t)$
DSB-SC	$m(t)$	0	$\frac{1}{2}\{m(t) + j0\}e^{j2\pi\Delta ft}$
DSB-WC	$1 + m(t)$	0	$\frac{1}{2}\{(1 + m(t)) + j0\}e^{j2\pi\Delta ft}$
SSB-SC	$m(t)$	$\pm\hat{m}(t)$	$\frac{1}{2}\{m(t) \pm j\hat{m}(t)\}e^{j2\pi\Delta ft}$
SSB-WC	$1 + m(t)$	$\pm\hat{m}(t)$	$\frac{1}{2}\{m(t) \pm j\hat{m}(t)\}e^{j2\pi\Delta ft}$

For FM modulation, the format is different since it requires angle modulation unlike other modulation types. A phase signal $\emptyset(t)$ is constructed from the message signal *m(t)*.

$$\emptyset(t) = 2\pi f_c t + 2\pi kf \int_{-\infty}^{t} m(t')dt'$$

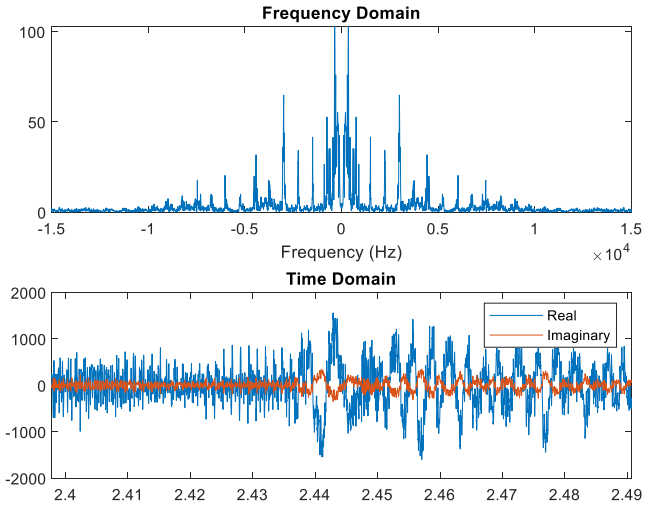
If we implement FM to I/Q modulation configuration, we obtain:

FM	$\cos(\emptyset(t))$	$sin(\emptyset(t))$	$\frac{1}{2}\{\cos(\emptyset(t)) + jsin(\emptyset(t))\}e^{j2\pi\Delta ft}$

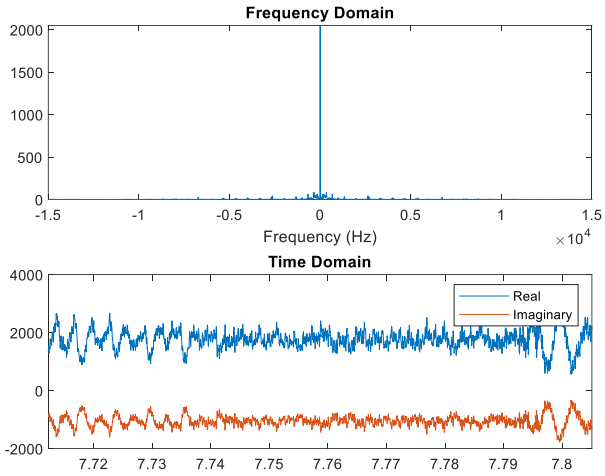
Although reference paper used GNU Radio for dataset generation [3], our approach was to utilize MATLAB[4] as the simulation tool for modulating an arbitrary message signal and implement the random impairments. A single sample of our dataset with the modulation types as described in Table 1 is a time domain signal ".mat" file consisting of 4096 samples with a fixed sample rate of 44.1 kHz.

22.01.2024

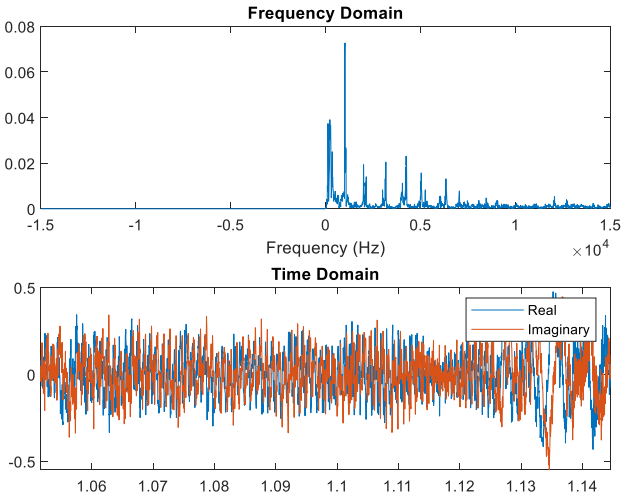
During the entire synthetic-waveform generation process, randomness is maintained by selecting the message in a random fashion, and regarding the channel impairments, selecting the CFO value and SNR in a random fashion.



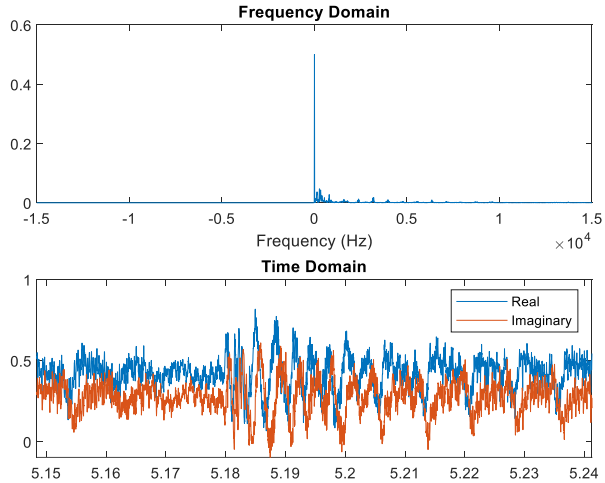
DSB-SC



DSB-WC

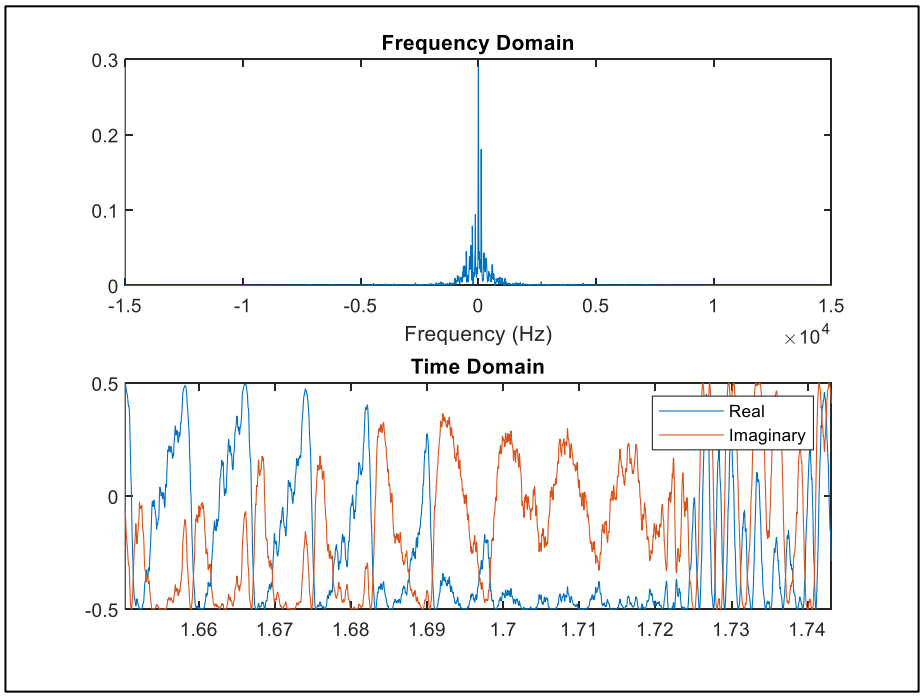


SSB-SC



SSB-WC

FM



4. Implementation of Deep Learning Architecture

Residual Network (ResNet) is a Deep Learning architecture we use to classify the modulation types of analog signals in this project. ResNet is introduced in [2] and it is widely used for image classification, object detection, and other tasks due to its effectiveness in training very deep neural networks. There are many types of ResNet model such as ResNet-18 which is mostly used in 2D image classification, and ResNet-34, used in medical imaging. The ResNet architecture we use in this project is custom ResNet. The name refers to a modified or tailored version of the standard ResNet architecture designed to suit a specific application or dataset. The usage of custom Resnet is determined by trial. In addition, we utilized Python's PyTorch library to train our model. Our model was ResNet1D.

To train our model, we firstly load created datasets – which has been explained in detail throughout the report- (.mat files) for each modulation and label each dataset by a number from 0 to 4.

Table 2: Labelling Dataset.

0	**1**	**2**	**3**	**4**
DSB-SC	DSB-WC	SSB-SC	SSB-WC	FM

After loading, we normalize of all loaded data by diving every data to its absolute maximum value. The dataset consists of complex numbers due to I/Q modulation. In order to utilize the dataset in a most efficient way, we represent the values as their magnitude and phase, and then combine this into a single tensor. This preprocessing step is done to convert our dataset to a configuration which can be utilized by dataset. After preprocessing the dataset is divided into three main parts which are training, validation and testing parts, in a balanced manner.

Moreover, we used stratified sampling here to preserve the proportion of each class. Since we created datasets in a way that there is same amount of data for each modulation, namely 4096 samples, the stratified sampling ensures that after splitting the dataset into 3 parts, the proportion between data numbers remains same. The training set is used for optimizing the model parameters, validation set is used for preventing overfitting which is the case when model learns the training data in a perfect way, it fails to infer about unseen data. The test set is used for evaluating model's performance on unseen data. Then, we convert our signals to PyTorch tensors to be able to use it in ResNet. Moreover, we added one more variable into tensors to make signals compatible with Conv1d layers. To load the data for batching we

Our ResNet model has two channels, one for the real an the other for imaginary part of the dataset. Furthermore, we have five number of classes which represents the different modulations schemes within the dataset. This part is crucial since labelling is the main idea behind training the model. We selected the kernel size conventionally by setting initial layers to 7 and 3 in residual blocks. These parameters were adjusted, and the performance of the classification was observed. As the final parameter, we set the class weights equal so that each modulation scheme is treated as the same during the training part of our residual network.

Table 3: ResNet Model Parameters.

Number of classes	5
Channels	2
Learning Rate	10^{-5}
Gamma	0.5
Weight Decay	10^{-4}
Step Size	10
Number of Epochs	4

5. Classification

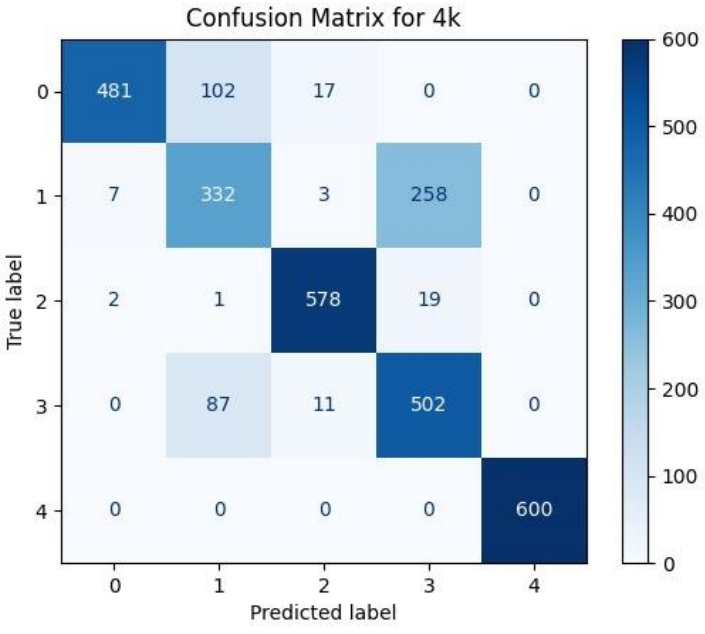


Figure 2 – Confusion Matrix

We generated 4k different fata for each modulation type and each data consist of 4096 samples. Then we fed this dataset to the Resnet model that we have described in Chapter 4.

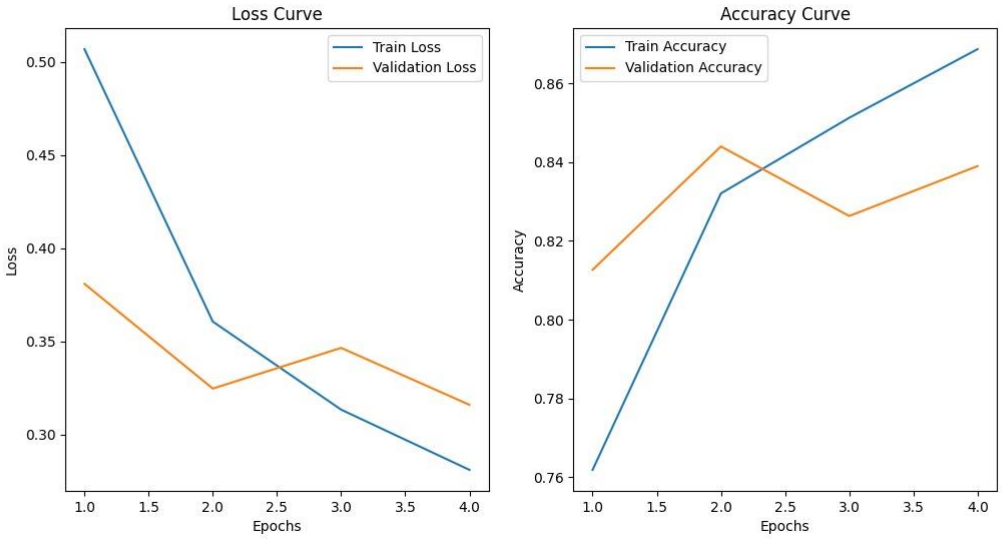


Figure 3 – Training Curves

After running our ResNet model on the generated datasets, we observed that in some cases the classification is flawless. We tried to implement the over-the-air transmission by using ADALM-PLUTO SDR. We saw that the accuracy of our model strictly depends on the number of samples selected to train the residual network. Moreover, we observed that number of epochs sometimes results in underfitting or overfitting depending on the training dataset. Since the Tx and Rx were very close to each other, there was no path loss effect observed.

Apart from generating the dataset correctly, another main challenge was to select residual network parameters in a proper fashion. Most of the time a trial-error approach was not feasible due to the computing time of the model. Moreover, fine-tuning was also required to obtain the desired accuracy of our model.

5. Conclusion

In this project, a deep learning-based classification system was implemented using ResNet to distinguish between various analog modulated signals. Synthetic datasets were generated for six modulation types, incorporating varying SNR and CFO conditions, to simulate realistic communication channel scenarios. The ResNet model demonstrated its effectiveness in accurately classifying the signals under these conditions, highlighting the potential of deep learning approaches in radio signal classification. The findings provide valuable insights into leveraging residual networks for signal analysis and lay the groundwork for further exploration with real-world signal scenarios and hardware implementations.

6. References

[1] T. J. O'Shea, T. Roy and T. C. Clancy, "Over-the-Air Deep Learning Based Radio Signal Classification," in IEEE Journal of Selected Topics in Signal Processing, vol. 12, no. 1, pp. 168-179, Feb. 2018, doi: 10.1109/JSTSP.2018.2797022.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," arXiv preprint arXiv:1512.03385, 2015. [Online]. Available: https://doi.org/10.48550/arXiv.1512.03385

[3] T. J. O'Shea and N. West, "Radio machine learning dataset generation with GNU radio," in Proc. GNU Radio Conf., 2016, vol. 1, no. 1.

[4] https://www.mathworks.com/help/comm/ug/modulation-classification-with-deep-learning.html

Appendix

Transmitter code

```
import matplotlib.pyplot as plt

from scipy.signal import butter, filtfilt

import sounddevice as sd

import adi

import numpy as np

from scipy.signal import butter, filtfilt

import torch

import torch.nn as nn

import torch.nn.functional as F

import iio

import time

class ResidualBlock(nn.Module):

    def __init__(self, in_channels, out_channels, stride=1, downsample=None):

        super(ResidualBlock, self).__init__()

        self.conv1 = nn.Conv1d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1,

bias=False)

        self.bn1 = nn.BatchNorm1d(out_channels)

        self.conv2 = nn.Conv1d(out_channels, out_channels, kernel_size=3, stride=1, padding=1,

bias=False)

        self.bn2 = nn.BatchNorm1d(out_channels)

        self.downsample = downsample


    def forward(self, x):

        residual = x

        if self.downsample is not None:
```

```
            residual = self.downsample(x)

        x = F.relu(self.bn1(self.conv1(x)))
        x = self.bn2(self.conv2(x))
        x += residual
        return F.relu(x)


class ResNet1D(nn.Module):
    def __init__(self, input_channels, num_classes):
        super(ResNet1D, self).__init__()
        self.initial_layer = nn.Sequential(
            nn.Conv1d(input_channels, 64, kernel_size=7, stride=2, padding=3, bias=False),
            nn.BatchNorm1d(64),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=3, stride=2, padding=1)
        )

        self.layer1 = self._make_layer(64, 64, 2, stride=1)
        self.layer2 = self._make_layer(64, 128, 2, stride=2)
        self.layer3 = self._make_layer(128, 256, 2, stride=2)
        self.layer4 = self._make_layer(256, 512, 2, stride=2)

        self.global_pool = nn.AdaptiveAvgPool1d(1)
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self, in_channels, out_channels, blocks, stride):
        downsample = None
        if stride != 1 or in_channels != out_channels:
            downsample = nn.Sequential(
                nn.Conv1d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm1d(out_channels)
            )
```

```
        layers = [ResidualBlock(in_channels, out_channels, stride, downsample)]
        for _ in range(1, blocks):
            layers.append(ResidualBlock(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.initial_layer(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.global_pool(x).squeeze(-1)
        x = self.fc(x)
        return x
```

```
# Configuration parameters
samp_rate = 2e6  # Sampling rate in Hz
rx_lo = 1e9  # Receiver LO frequency in Hz
rx_gain0 = 40  # Receiver gain
num_samples = 4096  # Number of samples
fs_audio = 44100  # Audio sampling rate in Hz

sdr = adi.Pluto('ip:192.168.2.1')

import numpy as np
import adi
import scipy.io
```

```
# Load the .mat file and extract the data
mat_file = 'alkm_ssbsc.mat'  # Replace with the path to your .mat file
mat_data = scipy.io.loadmat(mat_file)

# Assuming the variable 'data' in the .mat file is named 'data' in the dictionary
# Check the keys in the .mat file to ensure it's correctly named
print(mat_data.keys())

# Extract 'data' from the .mat file (replace 'data' with the correct key if necessary)
data = mat_data['data']

# Connect to the ADALM-PLUTO transmitter
# Set up transmission parameters
sdr.tx_lo = int(1e9)  # Local oscillator frequency
sdr.tx_gain = 0  # Attenuation/gain
sdr.tx_rf_bandwidth= int(200e3)

sdr.gain_control_mode_chan0 = "slow_attack"
# Enable cyclic buffers
sdr.tx_cyclic_buffer = True

# Set the sampling rate
sdr.sample_rate = int(samp_rate)

# Define message signal (assuming 'data' is a numpy array)
# Extract the 6th row (index 5) from the data variable
mesaj = data[5, :]  # Extract the 6th row (index 5)
print(mesaj)

# Prepare the signal for transmission
#i0 = mesaj * 2**14
#q0 = np.zeros_like(i0)  # Set the imaginary part (Q channel) to 0
#s_t = i0 + 1j * q0  # Complex signal (I + jQ)
s_t = mesaj.astype(np.complex64)  # Ensure correct type for transmission
```

```
# Send the data
sdr.tx(mesaj)
# Optionally, you can release the transmitter after usage
# tx.release()  # Uncomment if you want to explicitly release the device

time.sleep(1)

# Parameters
mean = 0
std_dev = 0.01

# Generate one sample
gaussian_sample = np.random.normal(loc=mean, scale=std_dev)

# Display the result
print("Gaussian Sample:", gaussian_sample)
```

```
sdr.rx_lo = int(rx_lo)
sdr.rx_hardwaregain_chan0 = rx_gain0  # Set gain
sdr.gain_control_mode_chan0 = "slow_attack"  # or "fast_attack"
sdr.rx_rf_bandwidth = int(200e3)
sdr.rx_buffer_size = int(num_samples)
sdr_data = sdr.rx()
```

```
# Connect to ADALM-PLUTO
sdr_data = sdr_data / np.max(np.abs(sdr_data))
```

```
# Capture data
data_received = np.abs(sdr_data)

# Normalize the received data
# data_received = data_received / np.max(np.abs(data_received))
received_signal = data_received
# Step 2: Define preprocessing function
def preprocess_signal(signal):
    """
    Preprocesses the signal to match the input requirements of the model.
    """
    signal = (signal - np.mean(signal)) / np.std(signal)  # Normalize
    signal = np.expand_dims(signal, axis=0)  # Add batch dimension
    signal = np.expand_dims(signal, axis=0)  # Add channel dimension
    return torch.tensor(signal, dtype=torch.float32)

# Step 3: Load the pre-trained model
def load_model(model_path, device):
    # Define model architecture
    input_channels = 1  # Adjust as per your data
    num_classes = 5  # Adjust as per your use case
    model = ResNet1D(input_channels, num_classes)

    # Load the state dictionary
    model.load_state_dict(torch.load(model_path, map_location=device, weights_only = True))
    model.to(device)
    model.eval()  # Set model to evaluation mode
    return model

# Step 4: Perform inference
def classify_modulation(model, processed_signal, device, modulation_classes):
    """
    Classifies the modulation type of the given signal.
    """
```

```
    processed_signal = processed_signal.to(device)
    with torch.no_grad():
        output = model(processed_signal)  # Perform inference
        predicted_label = torch.argmax(output, dim=1).item()  # Get the predicted label

    print(f"Predicted Modulation: {modulation_classes[predicted_label]}")
    return modulation_classes[predicted_label]

# Step 5: Main script to process and classify signal
if __name__ == "__main__":
    # Predefined parameters
    model_path = "outputs/final_model_4kdatawithcurve_ocak22.pth"  # Replace with your model path
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    modulation_classes = ["DSB-SC" ,"DSB-WC", "SSB-SC" ,"SSB-WC","FM"]  # Adjust as needed

    # Preprocess the received signal
    processed_signal = preprocess_signal(received_signal)

    # Load the pre-trained model
    model = load_model(model_path, device)

    # Classify the modulation type
    predicted_modulation = classify_modulation(model, processed_signal, device, modulation_classes)

    print(f"Final Predicted Modulation: {predicted_modulation}")
```

Resnet Model Code

```
import os
import numpy as np
import matplotlib.pyplot as plt
import scipy.io as sio
import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
```

```
import torch.nn.functional as F
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.model_selection import train_test_split


# ------------------------------
# 1. Load and Normalize Data
# ------------------------------
def load_mat_files(file_paths, labels):
    """Loads data from multiple .mat files and assigns labels."""
    all_signals = []
    all_labels = []

    for file_path, label in zip(file_paths, labels):
        data = sio.loadmat(file_path)
        signals = data['data']  # Replace with the correct key for your signal data
        all_signals.append(signals)
        all_labels.append(np.full((signals.shape[0],), label))
        #labels = data['label'].squeeze()  # Assuming 'label' is the correct key
        #print(f"File: {file_path}, Unique Labels: {np.unique(labels)}")

    return np.concatenate(all_signals, axis=0), np.concatenate(all_labels, axis=0)


def normalize_signals(signals):
    """Normalize each signal to zero mean and unit variance."""
    return (signals - signals.mean(axis=1, keepdims=True)) / signals.std(axis=1, keepdims=True)


# File paths and labels
file_paths = [
    r'C:\Users\EMIR\Desktop\comtermproject2nd stage\ocak_dsb_sc.mat',
    r'C:\Users\EMIR\Desktop\comtermproject2nd stage\ocak_22_dsb_wc.mat',
    r'C:\Users\EMIR\Desktop\comtermproject2nd stage\ocak_ssb_sc.mat',
```

```
    r'C:\Users\EMIR\Desktop\comtermproject2nd stage\ocak_ssb_wc.mat',
    r'C:\Users\EMIR\Desktop\comtermproject2nd stage\ocak_fm.mat'
]
labels = [0, 1, 2, 3, 4]

# Load and preprocess data
signals, labels = load_mat_files(file_paths, labels)
signals = normalize_signals(np.abs(signals))
print("Signals shape:", signals.shape)

# 2. Balanced Train/Validation/Test Splits
# ------------------------------
# Initialize lists for splits
train_signals_list, train_labels_list = [], []
val_signals_list, val_labels_list = [], []
test_signals_list, test_labels_list = [], []

# Get unique classes
unique_classes = np.unique(labels)

# Split data for each class
for cls in unique_classes:
    class_indices = labels == cls
    class_signals = signals[class_indices]
    class_labels = labels[class_indices]

    # Split into train and temp (validation + test)
    signals_train, signals_temp, labels_train, labels_temp = train_test_split(
        class_signals, class_labels, test_size=0.3, random_state=42, stratify=class_labels
    )

    # Split temp into validation and test
    signals_val, signals_test, labels_val, labels_test = train_test_split(
        signals_temp, labels_temp, test_size=0.5, random_state=42, stratify=labels_temp
```

```
    )

    # Append splits
    train_signals_list.append(signals_train)
    train_labels_list.append(labels_train)
    val_signals_list.append(signals_val)
    val_labels_list.append(labels_val)
    test_signals_list.append(signals_test)
    test_labels_list.append(labels_test)

# Combine splits
train_signals = np.concatenate(train_signals_list, axis=0)
train_labels = np.concatenate(train_labels_list, axis=0)
val_signals = np.concatenate(val_signals_list, axis=0)
val_labels = np.concatenate(val_labels_list, axis=0)
test_signals = np.concatenate(test_signals_list, axis=0)
test_labels = np.concatenate(test_labels_list, axis=0)

# Print distributions
print("Training Labels Distribution:", np.bincount(train_labels))
print("Validation Labels Distribution:", np.bincount(val_labels))
print("Testing Labels Distribution:", np.bincount(test_labels))
# ------------------------------
# 2. Define Dataset and DataLoaders
# ------------------------------
class SignalDataset(Dataset):
    def __init__(self, signals, labels):
        self.signals = torch.tensor(signals, dtype=torch.float32).unsqueeze(1)  # Add channel dimension
        self.labels = torch.tensor(labels, dtype=torch.long)

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
```

```
        return self.signals[idx], self.labels[idx]


# Split data into training, validation, and testing sets
'''train_size = int(0.7 * len(labels))
val_size = int(0.15 * len(labels))
test_size = len(labels) - train_size - val_size

train_signals, val_signals, test_signals = (
    signals[:train_size],
    signals[train_size:train_size + val_size],
    signals[train_size + val_size:]
)
train_labels, val_labels, test_labels = (
    labels[:train_size],
    labels[train_size:train_size + val_size],
    labels[train_size + val_size:]
)'''

# Create DataLoaders
train_dataset = SignalDataset(train_signals, train_labels)
val_dataset = SignalDataset(val_signals, val_labels)
test_dataset = SignalDataset(test_signals, test_labels)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# ------------------------------
# 3. Define ResNet1D Model
# ------------------------------
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(ResidualBlock, self).__init__()
```

```
        self.conv1 = nn.Conv1d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1,
bias=False)
        self.bn1 = nn.BatchNorm1d(out_channels)
        self.conv2 = nn.Conv1d(out_channels, out_channels, kernel_size=3, stride=1, padding=1,
bias=False)
        self.bn2 = nn.BatchNorm1d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        residual = x
        if self.downsample is not None:
            residual = self.downsample(x)

        x = F.relu(self.bn1(self.conv1(x)))
        x = self.bn2(self.conv2(x))
        x += residual
        return F.relu(x)


class ResNet1D(nn.Module):
    def __init__(self, input_channels, num_classes):
        super(ResNet1D, self).__init__()
        self.initial_layer = nn.Sequential(
            nn.Conv1d(input_channels, 64, kernel_size=7, stride=2, padding=3, bias=False),
            nn.BatchNorm1d(64),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=3, stride=2, padding=1)
        )

        self.layer1 = self._make_layer(64, 64, 2, stride=1)
        self.layer2 = self._make_layer(64, 128, 2, stride=2)
        self.layer3 = self._make_layer(128, 256, 2, stride=2)
        self.layer4 = self._make_layer(256, 512, 2, stride=2)
```

```
        self.global_pool = nn.AdaptiveAvgPool1d(1)
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self, in_channels, out_channels, blocks, stride):
        downsample = None
        if stride != 1 or in_channels != out_channels:
            downsample = nn.Sequential(
                nn.Conv1d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm1d(out_channels)
            )

        layers = [ResidualBlock(in_channels, out_channels, stride, downsample)]
        for _ in range(1, blocks):
            layers.append(ResidualBlock(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.initial_layer(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.global_pool(x).squeeze(-1)
        x = self.fc(x)
        return x



# ------------------------------
# 4. Compute Class Weights
# ------------------------------
unique_classes = np.unique(train_labels)
class_weights = compute_class_weight('balanced', classes=unique_classes, y=train_labels)
```

```
class_weights = torch.tensor(class_weights, dtype=torch.float32).to('cuda' if torch.cuda.is_available()
else 'cpu')
print("Class Weights:", class_weights)


# ------------------------------
# 5. Training and Validation
# ------------------------------
# ------------------------------
# 5. Training and Validation with Training Curves
# ------------------------------
def train_and_monitor(model, train_loader, val_loader, criterion, optimizer, scheduler, device,
num_epochs=10, output_dir="outputs"):
    best_val_accuracy = 0.0  # Initialize best validation accuracy

    # Initialize lists to store loss and accuracy values
    train_losses, val_losses = [], []
    train_accuracies, val_accuracies = [], []

    # Create output directory if it doesn't exist
    os.makedirs(output_dir, exist_ok=True)

    for epoch in range(num_epochs):
        model.train()
        train_loss = 0.0
        correct_train = 0
        total_train = 0

        # Training phase
        for signals, labels in train_loader:
            signals, labels = signals.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(signals)
            loss = criterion(outputs, labels)
```

```
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        _, preds = torch.max(outputs, 1)
        correct_train += (preds == labels).sum().item()
        total_train += labels.size(0)

    train_accuracy = correct_train / total_train
    train_loss /= len(train_loader)

    # Validation phase
    model.eval()
    val_loss = 0.0
    correct_val = 0
    total_val = 0

    with torch.no_grad():
        for signals, labels in val_loader:
            signals, labels = signals.to(device), labels.to(device)
            outputs = model(signals)
            loss = criterion(outputs, labels)

            val_loss += loss.item()
            _, preds = torch.max(outputs, 1)
            correct_val += (preds == labels).sum().item()
            total_val += labels.size(0)

    val_accuracy = correct_val / total_val
    val_loss /= len(val_loader)

    # Log loss and accuracy for this epoch
    train_losses.append(train_loss)
    val_losses.append(val_loss)
```

```
        train_accuracies.append(train_accuracy)
        val_accuracies.append(val_accuracy)

        # Print epoch statistics
        print(f"Epoch [{epoch + 1}/{num_epochs}], "
              f"Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.2%}, "
              f"Val Loss: {val_loss:.4f}, Val Accuracy: {val_accuracy:.2%}")

        # Save the model if it achieves the best validation accuracy
        if val_accuracy > best_val_accuracy:
            best_val_accuracy = val_accuracy
            torch.save(model.state_dict(), os.path.join(output_dir,
"best_model_4kdata_withcurve_ocak22.pth"))  # Save the model state dictionary
            print(f"New best model saved at epoch {epoch + 1} with Val Accuracy: {val_accuracy:.2%}")

        # Update learning rate scheduler
        scheduler.step()

    # Save the final model
    torch.save(model.state_dict(), os.path.join(output_dir, "final_model_4kdatawithcurve_ocak22.pth"))
    print(f"Training completed. Final model saved as 'final_model_4kdatawithcurve_ocak22.pth' in
{output_dir}.")

    # Plot and save training curves
    plot_training_curves(train_losses, val_losses, train_accuracies, val_accuracies, output_dir)

def plot_training_curves(train_losses, val_losses, train_accuracies, val_accuracies, output_dir):
    """Plots and saves the training and validation loss and accuracy curves."""
    epochs = range(1, len(train_losses) + 1)

    # Plot loss curves
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.plot(epochs, train_losses, label="Train Loss")
```

```
    plt.plot(epochs, val_losses, label="Validation Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title("Loss Curve")
    plt.legend()
    loss_curve_path = os.path.join(output_dir, "loss_curve_4kdata_ocak22.png")
    plt.savefig(loss_curve_path)
    print(f"Loss curve saved at {loss_curve_path}")

    # Plot accuracy curves
    plt.subplot(1, 2, 2)
    plt.plot(epochs, train_accuracies, label="Train Accuracy")
    plt.plot(epochs, val_accuracies, label="Validation Accuracy")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.title("Accuracy Curve")
    plt.legend()
    accuracy_curve_path = os.path.join(output_dir, "accuracy_curve_4kdata_ocak22.png")
    plt.savefig(accuracy_curve_path)
    print(f"Accuracy curve saved at {accuracy_curve_path}")

    plt.tight_layout()
    plt.show()


# ------------------------------
# 6. Evaluate and Visualize Confusion Matrix
# ------------------------------
def visualize_confusion_matrix(model, data_loader, device, num_classes, output_dir="outputs"):
    os.makedirs(output_dir, exist_ok=True)
    model.eval()
    all_preds = []
    all_labels = []
```

```
    with torch.no_grad():
        for signals, labels in data_loader:
            signals, labels = signals.to(device), labels.to(device)
            outputs = model(signals)
            _, preds = torch.max(outputs, 1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    cm = confusion_matrix(all_labels, all_preds, labels=list(range(num_classes)))
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=list(range(num_classes)))
    disp.plot(cmap=plt.cm.Blues)
    plt.title("Confusion Matrix for 4k")
    plt.savefig(os.path.join(output_dir, "confusion_matrix_4k_learningcurve_ocak22.png"))
    plt.show()

# ------------------------------
# 7. Run Training and Evaluation
# ------------------------------
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = ResNet1D(input_channels=1, num_classes=5).to(device)

criterion = nn.CrossEntropyLoss(weight=class_weights)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-4)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.5)
print("Training Labels Distribution:", np.bincount(train_labels))
print("Unique Training Labels:", np.unique(train_labels))
train_and_monitor(model, train_loader, val_loader, criterion, optimizer, scheduler, device,
num_epochs=4)
visualize_confusion_matrix(model, test_loader, device, num_classes=5, output_dir="outputs")
```