Applications of Machine Learning in Astrophysics: Computational Methods for Gravitational Wave Data Analysis, Classification, and Augmentation

Shufan Dong*

August 2024

Abstract

This paper provides a comprehensive overview of advanced methodologies for the analysis of gravitational wave (GW) data, emphasizing the integration of machine learning (ML) and deep learning (DL) techniques to enhance the detection and interpretation of GW signals. Initially, we discuss the foundational data preprocessing steps, including raw data acquisition, noise filtration, and data normalization, which are crucial for preparing GW datasets for ML applications. We then examine the fully preprocessed GW data with graphical information and statistical analysis, alongside a simplistic GW event classifier developed without ML applications. After that, to match the input data size of various ML models presented in this study, we detail the conversion of time-series GW data into spectrograms for 2D models like 2D CNNs, and the retention of time-series format for 1D and synthetic models: 1D CNNs; 1D RNNs, including Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU); and synthetic models, including Generative Adversarial Networks (GANs) and WaveNet. The study further explores the use of the following ML models for GW data analysis: Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), convolutional and recurrent autoencoders, Transformers, Deep Belief Networks (DBNs), Graph Neural Networks (GNNs), and synthetic models such as GANs and WaveNet. The analysis also includes the application of traditional ML models, such as Support Vector Machines (SVM), Random Forest Classifiers (RF), and Gaussian Mixture Models (GMM), providing a comparative evaluation of their effectiveness in classifying and detecting GW signals. Additionally, in the appendix section, we show a few examples of synthetic GW data generated using GANs and WaveNet models, offering a new potential to augment training datasets by improving model robustness with artificially

*Class of 2026, Bronx High School of Science, NY, USA.

synthesized GW data. Our results underline the significant potential of these methodologies in enhancing the accuracy and reliability of GW signal detection, thereby contributing to the broader field of astrophysical research.

Contents

1 Introduction

Gravitational wave (GW) astronomy has fundamentally transformed our understanding of the universe, offering a new insight through which we can observe cosmic phenomena that were previously impossible to access. These ripples in spacetime, first predicted by Albert Einstein in 1916 in his general theory of relativity, went undetected for nearly a century despite intense theoretical study. Nevertheless, the landscape of astrophysics changed dramatically in 2015 with the first direct detection of GWs from a binary black hole merger by the Laser Interferometer Gravitational-Wave Observatory (LIGO). This breakthrough not only confirmed a key prediction of Einstein's theory but also triggered the beginning of a new era in astrophysics—one in which the universe could be observed not only through electromagnetic (EM) waves but also through gravitational waves, offering a direct probe of celestial cosmic events.

This initial discovery was quickly followed by the detection of other high-energy astrophysical phenomena, such as neutron star mergers, which in 2017 provided a direct link between GWs and EM signals—a phenomenon termed "multi-messenger astronomy." These detections have been made possible by a global network of detectors, including LIGO in the United States, Virgo in Europe, and the KAGRA detector in Japan. Together, these instruments have allowed researchers to observe the universe in unprecedented ways, refining our understanding of the fundamental nature of gravity and contributing to fields such as cosmology, nuclear physics, and the study of celestial objects such as black holes and neutron stars.

However, GW signals are extremely faint, requiring the development of highly sophisticated data analysis techniques. The signals are often buried within substantial noise from environmental and instrumental sources, making the task of signal extraction particularly challenging. Traditional methods of data analysis, while effective, have been increasingly supplemented by machine learning (ML) and deep learning (DL) techniques. These techniques, with their ability to detect complex patterns in vast datasets, have revolutionized GW data processing. The integration of ML methods has significantly enhanced the accuracy, efficiency, and sensitivity of GW detection, allowing for the real-time identification of signals and the detailed analysis of their properties.

This paper aims to build on these advancements by exploring a range of ML methodologies applied to GW data analysis. Starting with essential data preprocessing steps—such as noise reduction, signal extraction, and data normalization—we prepare the groundwork for applying advanced ML models. The input data for each model is tailored to optimize its performance; for example, time-series data is typically used for models like 1D Recurrent Neural Networks (RNNs) and 1D Convolutional Neural Networks (CNNs), while spectrograms—capturing both time and frequency domain information—are suited for 2D CNNs and other models handling visual or sequential data. The different data representations allow each model to focus on specific aspects of the GW signal, such as temporal patterns or frequency-based features. These models include CNNs, RNNs (LSTMs and GRUs), convolutional and recurrent autoencoders, Transformers, Deep Belief Networks (DBNs), Graph Neural Networks (GNNs), generative adversarial networks (GANs), WaveNet, Support Vector Machines (SVM), Random Forest Classifiers (RF), and Gaussian Mixture Models (GMM). The focus is on optimizing the performance of these models for the detection, classification, and parameter estimation of GW signals.

In addition, the paper explores the potential of synthetic data generation using techniques like GANs and WaveNet, which provide augmented training datasets and improve model robustness using artificially generated GW signals created by these synthetic models. By enhancing training data through realistic simulations of GW signals, we can further improve the models' accuracy in identifying rare or complex GW events. As the field continues to evolve, these methods promise to push the boundaries of GW astronomy, enabling more detailed and insightful explorations of the universe's most violent and energetic processes.

2 Raw Data Preprocessing

2.1 Data Acquisition and Setup

2.1.1 Setting GPS Time and Detector

For this study, we focus on a specific GW event (GW150914, the first confirmed observation of GWs from colliding black holes).

```
# Set GPS time:
t_start = 1126259462.4
t_end = 1126259462.4 # For specific events, make t_end the same as t_start

# Choose detector (H1, L1, or V1)
detector = 'H1'
```

Figure 1: Locating GPS time for Binary Black Holes merger (BBH) event GW150914 and choosing the Hanford (H1) detector.

2.1.2 Importing TimeSeries Package

We ensure that we can successfully import TimeSeries from gwpy by installing the other required packages necessary for this installation.

```
try:
    from gwpy.timeseries import TimeSeries
except:
    ! pip install -q "gwpy==3.0.8"
    ! pip install -q "matplotlib==3.9.0"
    ! pip install -q "astropy==6.1.0"
    from gwpy.timeseries import TimeSeries
```

Figure 2: Importing TimeSeries from gwpy.

2.1.3 Downloading and Reading Data

The GW data is downloaded and read into a TimeSeries object.

```
from gwosc.locate import get_urls
url = get_urls(detector, t_start, t_end)[-1]

# If an event is chosen, then its info will be shown in url
print('Downloading: ' , url)
fn = os.path.basename(url)
with open(fn,'wb') as strainfile:
    straindata = requests.get(url)
    strainfile.write(straindata.content)
```

Figure 3: Downloading and reading the GW data with the TimeSeries package imported in the last subsection.

2.2 Data Extraction and Handling Missing Values

2.2.1 Extracting Data

The timestamps and strain values are extracted and stored in a pandas DataFrame.

```python
# Extract time and strain vals
timestamps = strain.times.value
strain_values = strain.value

# Store data in pd df
data = pd.DataFrame({
    'time': timestamps,
    'strain': strain_values
})
```

Figure 4: Extracting the time and strain features from the raw GW data file.

2.2.2 Handling Missing Values

Any missing values in the dataset are dropped to ensure clean data.

```python
# Drop rows with missing vals
data = data.dropna()

print("\nMissing vals after cleaning:")
print(data.isnull().sum())
```

Figure 5: Dropping any NaN values from the dataset.

2.3 Data Noise Filtering and Normalization

2.3.1 Band-Pass Filtering

Noise filtering is crucial in GW data analysis due to the presence of various noise sources that can distract us from the true signal. One common method is band-pass filtering, which allows signals within a specific frequency range to pass through while reducing the significance of signals outside this range. The low cutoff frequency (20 Hz) and high cutoff frequency (500 Hz) are chosen based on the expected characteristics of a BBH event. Consequently, applying a band-pass filter helps in enhancing the signal-to-noise ratio (SNR) of the GW data, increasing the exposure of the actual GW signal.

```python
# Band-pass filter function
def butter_bandpass(lowcut, highcut, fs, order=5):
    nyq = 0.5 * fs
    low = lowcut / nyq
    high = highcut / nyq
    b, a = butter(order, [low, high], btype='band')
    return b, a

def bandpass_filter(data, lowcut, highcut, fs, order=5):
    b, a = butter_bandpass(lowcut, highcut, fs, order=order)
    y = filtfilt(b, a, data)
    return y

# Filter params
lowcut = 20  # Low cutoff frequency (Hz)
highcut = 500  # High cutoff frequency (Hz)

# Band-pass filter strain data
data['strain'] = bandpass_filter(data['strain'], lowcut, highcut, 4096)
```

Figure 6: butter_bandpass function designs a band-pass filter with specified low and high cutoff frequencies, while bandpass_filter function applies the designed filter to the GW data, removing noise outside the specified frequency range.

2.3.2 Data Normalization

Normalization is another crucial preprocessing step that adjusts the GW data to a common scale, making it easier to analyze and compare. This step ensures that the strain data have a mean of zero and a standard deviation of one. Standardizing the strain data is essential for ensuring that all features contribute equally to the analysis and for improving the performance of ML models that are sensitive to the scale of the data.

```python
# Normalize strain data
scaler = StandardScaler()
data['strain'] = scaler.fit_transform(data[['strain']])
```

Figure 7: StandardScaler function standardizes the features so that they're easier for ML algorithms to analyze.

2.4 Final Data Inspection

We briefly look at the data after it's being preprocessed.

```
First few rows of data:
           time    strain
0  1.126257e+09 -2.509170
1  1.126257e+09  0.070279
2  1.126257e+09  2.209691
3  1.126257e+09  3.618610
4  1.126257e+09  4.256309

Col headers:
Index(['time', 'strain'], dtype='object')

Summary stats:
               time        strain
count  1.677722e+07  1.677722e+07
mean   1.126259e+09 -1.758737e-17
std    1.182413e+03  1.000000e+00
min    1.126257e+09 -3.686864e+00
25%    1.126258e+09 -7.088868e-01
50%    1.126259e+09  1.167451e-03
75%    1.126260e+09  7.087773e-01
max    1.126262e+09  4.284804e+00

Missing vals in each col:
time      0
strain    0
dtype: int64
Sampling frequency: 4096.0 Hz Hz
```

Figure 8: Characteristics and features of the preprocessed GW data.

3 Data Visualization and Analysis

Visualization is an essential tool in GW data analysis, offering clear insights into the behavior and structure of astrophysical sources. In the time domain, GW features reveal key dynamics of compact objects like black holes and neutron stars, such as their masses, spins, and orbital characteristics. Time-domain analysis also highlights transient events, like mergers, and plays a crucial role in identifying noise to improve the signal-to-noise (SNR) ratio. Traditional, simplistic event detection focuses on recognizing significant signals from astrophysical phenomena, enabling timely follow-up observations across multiple observatories, and supporting multi-messenger astronomy. Lastly, parameter estimation determines the physical attributes of GW sources, allowing for rigorous tests of gravitational theories and enhancing our understanding of the population and evolution of compact celestial objects.

3.1 Time Series Plot

We visualize how the strain data changes over time.

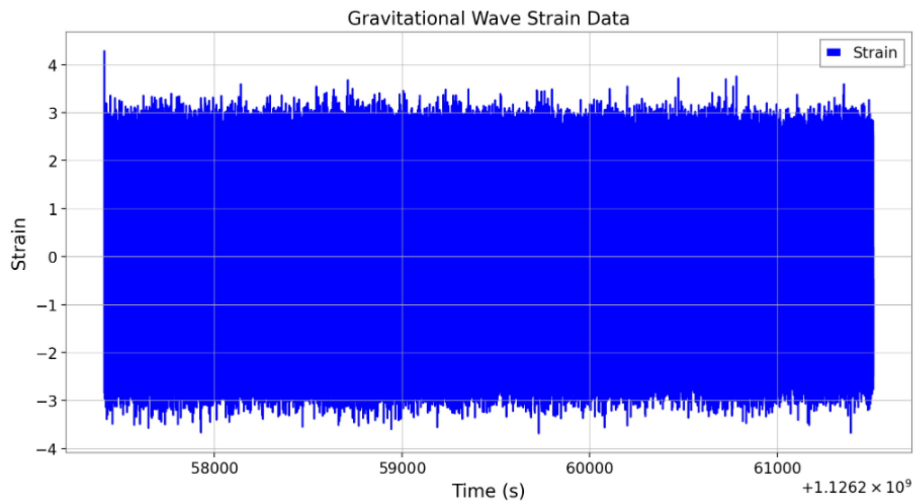

Figure 9: Graph of time-series plot (strain data versus time).

In the plot, peaks and troughs may correspond to significant events such as black hole mergers or neutron star collisions, and it is useful for initial data inspection, allowing us to identify the presence of potential GW events.

3.2 Spectrogram

We visualize how the frequency content of the strain data changes over time.

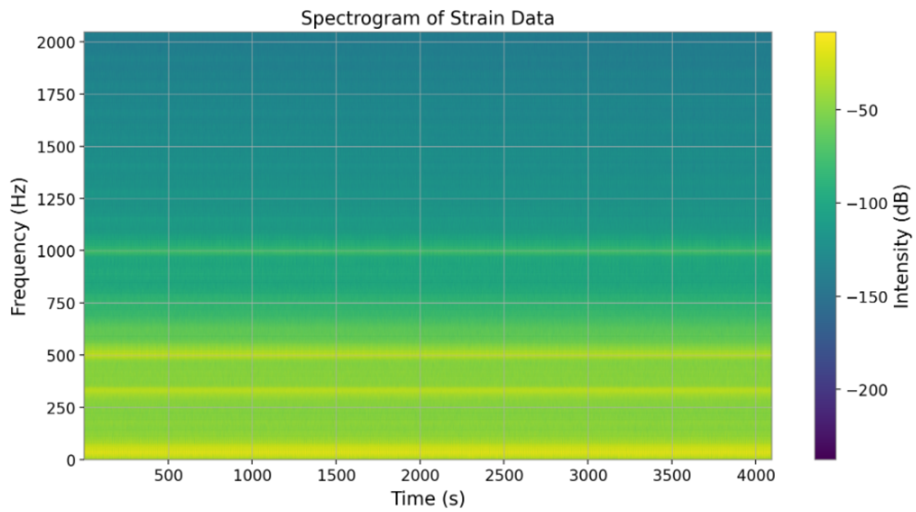

Figure 10: Graph of spectrograms (strain data's frequency versus time).

This plot helps identify transient events and their frequency components, which are crucial for distinguishing between noises and actual GW signals. Additionally, spectrograms provides a detailed view of how the signal's frequency content evolves, and spectrogram data can be used as 2D GW data for the implementation of certain ML models.

3.3 Histogram

We visualize the distribution of strain values.

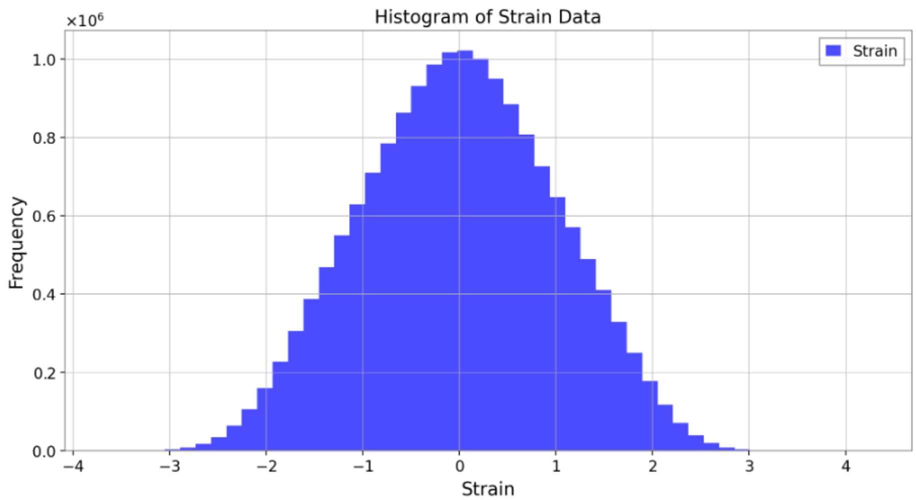

Figure 11: Graph of Histogram (frequency distribution of strain data).

This plot provides an overview of the data's spread, central tendency, and outliers. This is useful for identifying any anomalies or patterns in the data. Besides this, understanding the distribution of the strain values is crucial for subsequent statistical analysis and for ensuring that the GW data meets the expectations of various ML algorithms.

3.4 Time-Domain Features

The function `calc_and_print_time_domain_features` is designed to extract and print key time-domain features from GW data.

```python
def calc_and_print_time_domain_features(data, strain_column, fs):
    peak_amplitude = np.max(data[strain_column])
    min_amplitude = np.min(data[strain_column])
    print(f"Peak Amplitude ({strain_column}): {peak_amplitude}")
    print(f"Min Amplitude ({strain_column}): {min_amplitude}")

    threshold = 0.5 * peak_amplitude
    significant_signal = data[strain_column].abs() > threshold
    signal_duration = significant_signal.sum() * (1/fs)
    print(f"Signal Duration ({strain_column}): {signal_duration}s")

    signal_power = np.mean(data[strain_column]**2)
    noise_power = np.mean(data[data[strain_column].abs() <= threshold][strain_column]**2)
    snr = 10 * np.log10(signal_power / noise_power)
    print(f"Signal-to-Noise Ratio (SNR) ({strain_column}): {snr} dB\n")

# Calc features for strain data
print("Calc features for strain data: ")
calc_and_print_time_domain_features(data, 'strain', fs)
```

Figure 12: The function accepts three parameters: data (a DataFrame containing the signal and time data), strain_column (the strain data column), and fs (the sampling frequency), and the function calculates the peak and minimum amplitudes of the specified strain column. For computational purposes, a threshold is set at 50% of the peak amplitude, and the duration of significant signals exceeding this threshold is calculated and printed. As a result, the function calculates and prints the signal power, noise power, and SNR.

Time-domain features in GW data are crucial because they provide direct insights into the dynamics of astrophysical sources and the propagation of GWs. By analyzing these features, we can extract critical information about the nature and behavior of compact objects, such as black holes and neutron stars, and the environments in which they reside.

For instance, the shape and structure of the GW signal in the time domain can reveal the mass, spin, and orbital dynamics of a binary merger event. Features such as chirps, where the frequency and amplitude of the wave increase as the objects spiral closer, are particularly informative. Also, detecting short-lived, transient signals helps identify specific events like black hole mergers and neutron star collisions, and each of them has a unique, discoverable signature in the time domain.

Time-domain analysis allows for the identification of noises, which is essential for improving the signal-to-noise ratio (SNR) and ensuring the accuracy of the detected signals.

```
Calc features for strain data:
Peak Amplitude (strain): 4.284804453733104
Min Amplitude (strain): -3.686864089465157
Signal Duration (strain): 92.680908203125s
Signal-to-Noise Ratio (SNR) (strain): 0.4926804961051281 dB
```

Figure 13: The output of the function prints the peak amplitude, minimum amplitude, signal duration, and SNR.

3.5 Basic Event Detection and Parameter Estimation

The `calc_threshold` function calculates a threshold for event detection based on the standard deviation of the noise in the strain data.

```
def calc_threshold(data, strain_column, factor=3):
    noise_std = np.std(data[strain_column])
    threshold = factor * noise_std
    return threshold
```

Figure 14: This function calculates a threshold based on the standard deviation of the strain data. The threshold is set to a multiple of this standard deviation. A threshold of approximately 3 is calculated and returned.

The `detect_events` function identifies events in the strain data based on the calculated threshold.

```python
def detect_events(data, strain_column, threshold):
    events = []
    event_start = None

    for i, strain in enumerate(data[strain_column]):
        if abs(strain) > threshold:
            if event_start is None:
                event_start = i
        else:
            if event_start is not None:
                event_end = i
                events.append((event_start, event_end))
                event_start = None

    # Check if an event is ongoing at end of data
    if event_start is not None:
        events.append((event_start, len(data[strain_column]) - 1))
    return events
```

Figure 15: This function identifies events where the absolute strain exceeds the calculated threshold, and it iterates through the strain data, marking the start and end of events. In the end, detected events are stored as start and end indices in a list.

Event detection is the process of identifying important signals within the GW data that correspond to astrophysical phenomena. Rapid detection enables follow-up observations with EM and other observatories, providing critical support to multi-messenger astronomy.

The `estimate_event_params` function calculates parameters for each detected event.

```python
def estimate_event_params(data, strain_column, events, fs):
    time_column = 'time'

    event_params = []
    for event in events:
        start_idx, end_idx = event
        event_data = data[strain_column].iloc[start_idx:end_idx]
        peak_amplitude = np.max(np.abs(event_data))
        duration = (end_idx - start_idx) / fs
        event_params.append({
            'start_time': data[time_column].iloc[start_idx],
            'end_time': data[time_column].iloc[end_idx - 1],
            'peak_amplitude': peak_amplitude,
            'duration': duration
        })
    return event_params
```

Figure 16: This function calculates parameters such as start time (GPS time), end time (GPS time), peak amplitude, and duration for each detected event. For each event, the function extracts relevant data and calculates the required parameters, storing them in an array.

Parameter estimation conveys the importance of determining the physical parameters of the GW source, such as masses, spins, distances, and orbital characteristics. Accurate parameter estimation is vital for interpreting GW observations and understanding the underlying physics.

High-precision parameter estimation allows for stringent tests of general relativity and other gravitational theories. Detailed parameter estimation helps expound the population properties of compact objects, their formation channels, and their role in the cosmos.

```
Event Params:
{'start_time': 1126257415.0007324, 'end_time': 1126257415.001709, 'peak_amplitude': 4.284804453733104, 'duration': 0.001220703125}
{'start_time': 1126257418.3012695, 'end_time': 1126257418.3015137, 'peak_amplitude': 3.10656720831358, 'duration': 0.00048828125}
{'start_time': 1126257418.4667969, 'end_time': 1126257418.467041, 'peak_amplitude': 3.1396660570114685, 'duration': 0.00048828125}
{'start_time': 1126257423.3283691, 'end_time': 1126257423.3283691, 'peak_amplitude': 3.0493328722819033, 'duration': 0.000244140625}
{'start_time': 1126257423.4399414, 'end_time': 1126257423.4401855, 'peak_amplitude': 3.139458428439673, 'duration': 0.00048828125}
{'start_time': 1126257423.4418945, 'end_time': 1126257423.4421387, 'peak_amplitude': 3.1423561734113865, 'duration': 0.00048828125}
{'start_time': 1126257424.4831543, 'end_time': 1126257424.4833984, 'peak_amplitude': 3.0957766543715897, 'duration': 0.00048828125}
{'start_time': 1126257424.4851074, 'end_time': 1126257424.4855957, 'peak_amplitude': 3.1352130176788724, 'duration': 0.000732421875}
{'start_time': 1126257424.4865723, 'end_time': 1126257424.4873047, 'peak_amplitude': 3.198057213381449, 'duration': 0.0009765625}
{'start_time': 1126257426.8842773, 'end_time': 1126257426.8845215, 'peak_amplitude': 3.186333462298193, 'duration': 0.00048828125}
```

Figure 17: These are the event parameters of the first 10 events detected.

3.6 Basic Statistical Analysis

The `summarize_event_params` function summarizes the parameters of detected events.

```python
def summarize_event_params(event_params):
    if not event_params:  # Check if event_params array is empty
        return {
            'num_events': 0,
            'average_duration': 0,
            'max_duration': 0,
            'average_peak_amplitude': 0,
            'max_peak_amplitude': 0
        }

    durations = [param['duration'] for param in event_params]
    peak_amplitudes = [param['peak_amplitude'] for param in event_params]

    summary = {
        'num_events': len(event_params),
        'average_duration': np.mean(durations),
        'max_duration': np.max(durations),
        'average_peak_amplitude': np.mean(peak_amplitudes),
        'max_peak_amplitude': np.max(peak_amplitudes)
    }
    return summary
```

Figure 18: This function summarizes detected event parameters, and if no events are detected, it returns a summary with zeros. For detected events, it calculates and returns the number of events, average duration, maximum duration, average peak amplitude, and maximum peak amplitude.

```
Summary of Event Params:
{'num_events': 1645, 'average_duration': 0.0005074266242401216, 'max_duration': 0.004150390625, 'average_peak_amplitude': 3.127237008782134, 'max_peak_amplitude': 4.284804453733104}
```

Figure 19: This is the summary of the detected events and their corresponding parameters, including total number of events detected, average duration, maximum duration, average peak amplitude, and maximum peak amplitude.

4 Data Preparation and Augmentation

4.1 Data Segmentation and Labeling

The continuous GW strain data is split into smaller, manageable segments and labeled appropriately. This step is critical for preparing the dataset for supervised learning, allowing the model to learn based on discoverable patterns.

```python
def create_segments_and_labels(strain, event_time, window_size, sample_rate):
    # Resample strain to desired sample rate (if necessary)
    strain = strain.resample(sample_rate)

    # Def segments and labels ls
    segments = []
    labels = []

    # Calc # of samples per segment
    segment_length = int(window_size * sample_rate)

    # Create segments and labels
    for i in range(0, len(strain) - segment_length, segment_length):
        segment = strain[i:i + segment_length]
        segments.append(segment.value)

        # Label based on event presence
        if segment.times.value[0] <= event_time <= segment.times.value[-1]:
            labels.append(1)  # Event present
        else:
            labels.append(0)  # No event

    # Convert to np arrays
    segments = np.array(segments)
    labels = np.array(labels)

    return segments, labels

segments, labels = create_segments_and_labels(strain, t_start, 2, fs)
```

Figure 20: The function `create_segments_and_labels` is used to split the strain data into segments of 2 seconds each, starting at `t_start` (start of GW150914 event) and sampled at `fs` Hz (4096 Hz).

```
Segments shape: (2047, 8192)
Labels shape: (2047,)
```

Figure 21: The shape of GW data's segments and labels.

4.2 Time-series Data Reshaping for 1D and Synthetic Models

To ensure the compatibility of the time-series data for 1D and synthetic models, time-series data is reshaped to include an extra dimension.

```
# Reshape segments for 1D CNN
segments = segments.reshape((segments.shape[0], segments.shape[1], 1))

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(segments, labels, test_size=0.2, random_state=42)
```

Figure 22: The segment data is reshaped with an additional dimension of 1. Then, the data is split into the training set (80% of the data) and the testing set (20% of the data).

```
Reshaped segments shape: (2047, 8192, 1)
```

Figure 23: The shape of the input time-series data.

4.3 Spectrogram Data Generation for 2D Models

To examine the spatial feature extraction capabilities of 2D models, time-series data is converted into spectrograms, which provide a frequency domain representation of the data.

```
# Generate spectrograms for each segment
def generate_spectrogram(segment, sample_rate):
    f, t, Sxx = spectrogram(segment, sample_rate)
    return Sxx

spectrograms = np.array([generate_spectrogram(segment, fs) for segment in segments])

# Reshape spectrograms for 2D CNN
spectrograms = spectrograms[..., np.newaxis]  # Add a channel dim

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(spectrograms, labels, test_size=0.2, random_state=42)
```

Figure 24: The `generate_spectrogram` function converts each time-series segment into a spectrogram, and the spectrograms are then reshaped to include a channel dimension for compatibility with 2D model input. Then, the data is split into the training set (80% of the data) and testing set (20% of the data).

```
Reshaped spectrograms shape: (2047, 129, 36, 1)
```

Figure 25: The shape of the input spectrogram data.

4.4 Dataloader Generation for Transformer

Data preparation for the Transformer model involves creating a custom, plain dataset class used to convert the data into PyTorch tensors and using PyTorch's DataLoader for batching, shuffling, and splitting.

```python
class GWDataset(Dataset):
    def __init__(self, segments, labels):
        self.segments = segments
        self.labels = labels

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

    def __getitem__(self, idx):
        segment = torch.tensor(self.segments[idx], dtype=torch.float32)
        label = torch.tensor(self.labels[idx], dtype=torch.long)
        return segment, label

# Create dataset and dataloader
dataset = GWDataset(segments_aug, labels_aug)
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [train_size, test_size])

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Figure 26: For batching and shuffling purposes, the data is split into training (80%) and testing (20%) datasets using Python functions and PyTorch.

4.5 Tensor Data Creation for DBN

For the DBN model, the data is split into training and testing sets using Scikit-Learn's train_test_split function, and it's then converted to PyTorch tensors.

```
# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(segments, labels, test_size=0.2, random_state=42)

X_train_aug, y_train_aug = augment_data(X_train, y_train)

# Pytorch tensors
X_train_aug = torch.tensor(X_train_aug, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train_aug = torch.tensor(y_train_aug, dtype=torch.float32).view(-1, 1)
y_test = torch.tensor(y_test, dtype=torch.float32).view(-1, 1)
```

Figure 27: The data here is split into training (80%) and testing (20%) datasets with simply the Scikit-Learn's train_test_split function.

4.6 Graphical Data Generation for GNN

Graph-structured data is created for the GNN model, which captures complex relationships and structures in the GW data.

```
# Graph data
def create_graph_data(gw_signals, labels):
    graphs = []
    for signal, label in zip(gw_signals, labels):
        node_features = torch.tensor(signal, dtype=torch.float).unsqueeze(1)
        edge_index = torch.tensor([[i, i+1] for i in range(len(signal)-1)], dtype=torch.long).t().contiguous()
        y = torch.tensor([label], dtype=torch.long)
        graph = Data(x=node_features, edge_index=edge_index, y=y)
        graphs.append(graph)
    return graphs

graph_data = create_graph_data(segments_aug, labels_aug)

# Dataloader
data_loader = DataLoader(graph_data, batch_size=256, shuffle=True)
```

Figure 28: For its spatial capturing capabilities, GNN requires graphical data input, and PyTorch's DataLoader is utilized for batching and shuffling

Loop Over Signals and Labels

- The `zip(gw_signals, labels)` function pairs each signal with its corresponding label.
- `torch.tensor(signal, dtype=torch.float)`: converts the signal into a PyTorch tensor.
- `.unsqueeze(1)`: adds an extra dimension to the tensor.

- `[[i, i+1] for i in range(len(signal)-1)]`: creates pairs of consecutive indices (i, i+1), representing the edges between consecutive nodes in the graph.
- `torch.tensor(..., dtype=torch.long)`: converts index pairs into a PyTorch tensor.
- `.t()`: transposes the tensor.
- `.contiguous()`: ensures that the tensor's memory layout is compatible for efficient processing.
- `torch.tensor([label], dtype=torch.long)`: converts the label into a PyTorch tensor.
- `Data(x=node_features, edge_index=edge_index, y=y)`: creates a graph data object using the Data class from PyTorch Geometric.

The function at the end returns the list of graph data objects.

4.7 Data Augmentation

To prevent overfitting and improve generalization, data augmentation techniques are applied to the training data.

```
def augment_data(data, labels):
    augmented_data = []
    augmented_labels = []
    for d, l in zip(data, labels):
        augmented_data.append(d)
        augmented_labels.append(l)
        augmented_data.append(np.flip(d, axis=0))
        augmented_labels.append(l)
        noise = np.random.normal(0, 0.1, d.shape)
        augmented_data.append(d + noise)
        augmented_labels.append(l)
    return np.array(augmented_data), np.array(augmented_labels)

X_train_aug, y_train_aug = augment_data(X_train, y_train)
```

Figure 29: The `augment_data` function artificially increases the size of the training dataset by introducing variability.

5 Model Building, Training, and Evaluation

5.1 CNNs and RNNs

CNNs and RNNs are key architectures in DL, designed for different types of data. CNNs, especially 2D CNNs, are highly effective for spatial data, like images, using convolutional layers to detect patterns like edges and textures, making them ideal for image classification. RNNs excel in handling sequential data, such as time series data, by using loops to maintain context across input sequences, making them suitable for time-series prediction. These two types of DL models are commonly utilized for binary event classification in GW research.

5.1.1 1D CNN

A 1D CNN model is constructed and trained on the augmented time-series data.

```
Layer (type)                Output Shape              Param #
=================================================================
conv1d (Conv1D)             (None, 8190, 16)          64

max_pooling1d (MaxPooling1  (None, 4095, 16)          0
D)

conv1d_1 (Conv1D)           (None, 4093, 32)          1568

max_pooling1d_1 (MaxPoolin  (None, 2046, 32)          0
g1D)

flatten_1 (Flatten)         (None, 65472)             0

dense_2 (Dense)             (None, 64)                4190272

dropout_1 (Dropout)         (None, 64)                0

dense_3 (Dense)             (None, 1)                 65

=================================================================
```

Figure 30: The 1D CNN model processes the time-series data directly, using convolutional layers to extract temporal features, pooling layers to reduce dimensionality, dense layers to classify event presence, and a dropout layer to prevent overfitting.

5.1.2 2D CNN

A 2D CNN model is built and trained on the augmented spectrogram data.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 127, 34, 16)	160
max_pooling2d (MaxPooling2D)	(None, 63, 17, 16)	0
conv2d_1 (Conv2D)	(None, 61, 15, 32)	4640
max_pooling2d_1 (MaxPooling2D)	(None, 30, 7, 32)	0
flatten (Flatten)	(None, 6720)	0
dense (Dense)	(None, 64)	430144
dropout (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 1)	65

Figure 31: The 2D CNN model consists of convolutional layers for feature extraction, pooling layers for dimensionality reduction, and dense layers for event classification. A dropout layer is added to help prevent overfitting.

5.1.3 LSTM

An LSTM model is constructed and trained on the augmented time-series data.

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 2048, 64)	16896
dropout (Dropout)	(None, 2048, 64)	0
lstm_1 (LSTM)	(None, 64)	33024
dropout_1 (Dropout)	(None, 64)	0
dense (Dense)	(None, 1)	65

Figure 32: The LSTM model processes the time-series data directly, using two LSTM layers for feature extraction, two dropout layers to prevent overfitting, and a dense layer to classify event presence. For quicker model training, the size of the data for LSTM is resampled to four times less than the data for 1D and 2D CNN

5.1.4 GRU

A GRU model is constructed and trained on the augmented time-series data.

Layer (type)	Output Shape	Param #
gru (GRU)	(None, 2048, 64)	12864
dropout_2 (Dropout)	(None, 2048, 64)	0
gru_1 (GRU)	(None, 64)	24960
dropout_3 (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 1)	65

Figure 33: The GRU model processes the time-series data directly, using two GRU layers for feature extraction, two dropout layers to prevent overfitting, and a dense layer to classify event presence. For quicker model training, the size of the data for LSTM is resampled to four times less than the data for 1D and 2D CNN

5.2 Autoencoders

In addition to the application of convolutional and recurrent layers, the primary purpose of the autoencoders is to first compress the dimensions of the data in the encoder section and then expand the dimensions back in the decoder section, with the bottleneck section in the middle to mark the end of data dimensionality reduction and the start of data dimensionality expansion, and this is similar to as if you are to visualize the Big Bounce hypothesis on the contraction and expansion of the universe. Because of the unique training process of these autoencoders, ReLU activation is chosen for its non-linearity. Additionally, this method attempts to reconstruct the original input at the end of the training process, and then we can visualize how well the autoencoder performs at this reconstruction step to determine its ability in GW event detection.

5.2.1 1D CNN Autoencoder

1D CNN Autoencoder is efficient in extracting temporal features from time-series data.

```python
# Encoder
encoder = Sequential([
    Conv1D(16, 3, activation='relu', input_shape=(segments.shape[1], 1)),
    MaxPooling1D(2),
    Conv1D(32, 3, activation='relu'),
    MaxPooling1D(2),
    Conv1D(64, 3, activation='relu'),
    MaxPooling1D(2)
])

# Bottleneck
bottleneck = Sequential([
    Flatten(),
    Dense(32, activation='relu')
])

# Decoder
decoder = Sequential([
    Dense(64 * (segments.shape[1] // 8), activation='relu', input_shape=(32,)),
    Reshape((segments.shape[1] // 8, 64)),
    UpSampling1D(2),
    Conv1D(32, 3, activation='relu', padding='same'),
    UpSampling1D(2),
    Conv1D(16, 3, activation='relu', padding='same'),
    UpSampling1D(2),
    Conv1D(1, 3, activation='sigmoid', padding='same')
])
```

Figure 34: The 1D CNN autoencoder contains an encoder section (with 1D convolutional layers for feature extraction and 1D pooling layers for spatial dimensionality reduction), a bottleneck section (with a flatten layer to convert the data from 1D feature maps into a 1D vector and a dense layer for dimensionality reduction), and a decoder section (with a dense layer to expands the compressed data into higher dimensional space, a reshape layer to map the data from 1D vector to 2D tensor, 1D convolutional layers to feature refining, and 1D upsampling layers for dimensionality expansion).

5.2.2 2D CNN Autoencoder

2D CNN autoencoder is effective in capturing spatial hierarchies from spectrograms.

```
# Encoder
encoder = Sequential([
    Conv2D(16, (3, 3), activation='relu', padding='same', input_shape=(spectrograms.shape[1], spectrograms.shape[2], 1)),
    MaxPooling2D((2, 2), padding='same'),
    Conv2D(32, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2), padding='same'),
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2), padding='same')
])

# Bottleneck
bottleneck = Sequential([
    Flatten(),
    Dense(256, activation='relu')
])

# Decoder
decoder = Sequential([
    Dense(64 * (spectrograms.shape[1] // 8) * (spectrograms.shape[2] // 8), activation='relu', input_shape=(256,)),
    Reshape(((spectrograms.shape[1] // 8), (spectrograms.shape[2] // 8), 64)),
    UpSampling2D((2, 2)),
    Conv2D(32, (3, 3), activation='relu', padding='same'),
    UpSampling2D((2, 2)),
    Conv2D(16, (3, 3), activation='relu', padding='same'),
    UpSampling2D((2, 2)),
    Conv2D(1, (3, 3), activation='sigmoid', padding='same')
])
```

Figure 35: The 2D CNN autoencoder contains an encoder section (with 2D convolutional layers for feature extraction and 2D pooling layers for spatial dimensionality reduction), a bottleneck section (with a flatten layer to map the data from 2D feature maps into a 1D vector and a dense layer for dimensionality reduction), and a decoder section (with a dense layer to expands the compressed data into higher dimensional space, a reshape layer to map the data from 1D vector to 3D vector, 2D convolutional layers to feature refining, and 2D upsampling layers for dimensionality expansion).

5.2.3 LSTM Autoencoder

LSTM Autoencoder captures and learns long-term dependencies in sequential data.

```python
# Encoder
encoder = Sequential([
    LSTM(64, activation='relu', return_sequences=True, input_shape=(segments.shape[1], 1)),
    LSTM(32, activation='relu', return_sequences=False)
])

# Bottleneck
bottleneck = Sequential([
    Dense(32, activation='relu')
])

# Decoder
decoder = Sequential([
    RepeatVector(segments.shape[1]),
    LSTM(32, activation='relu', return_sequences=True),
    LSTM(64, activation='relu', return_sequences=True),
    TimeDistributed(Dense(1))
])
```

Figure 36: The LSTM autoencoder contains an encoder section (with LSTM layers for data processing and timesteps returning), a bottleneck section (with a dense layer for dimensionality reduction), and a decoder section (with a RepeateVector layer to simply repeat the compressed data for it to match the input sequence length, LSTM layers to preprocess the data for the repeated vector and return its timesteps, a TimeDistributed layer to apply a dense layer to each timestep to reconstruct the original input data).

5.2.4 GRU Autoencoder

GRU Autoencoder is efficient memory usage and effective for sequential dependencies.

```
# Encoder
encoder = Sequential([
    GRU(64, activation='relu', return_sequences=True, input_shape=(segments.shape[1], 1)),
    GRU(32, activation='relu', return_sequences=False)
])

# Bottleneck
bottleneck = Sequential([
    Dense(32, activation='relu')
])

# Decoder
decoder = Sequential([
    RepeatVector(segments.shape[1]),
    GRU(32, activation='relu', return_sequences=True),
    GRU(64, activation='relu', return_sequences=True),
    TimeDistributed(Dense(1))
])
```

Figure 37: The GRU autoencoder contains an encoder section (with GRU layers for data processing and timesteps returning), a bottleneck section (with a dense layer for dimensionality reduction), and a decoder section (with a RepeateVector layer to simply repeat the compressed data for it to match the input sequence length, GRU layers to preprocess the data for the repeated vector and return its timesteps, a TimeDistributed layer to apply a dense layer to each timestep to reconstruct the original input data).

5.3 Transformer

A Transformer model is defined and trained for time-series data classification, utilizing its ability to capture long-range dependencies in the data.

```python
# Set hyperparams
input_dim = 1 # time-series data
model_dim = 128
num_heads = 8
num_layers = 4
lr = 1e-4
batch_size = 256
dropout_rate = 0.2
output_dim = 2 # binary classificiation of event presence
num_epochs = 5
```

Figure 38: All the hyperparameters needed to train the Transformer model.

```python
class TransformerModel(nn.Module):
    def __init__(self, input_dim, model_dim, num_heads, num_layers, dropout_rate, output_dim):
        super(TransformerModel, self).__init__()
        self.embedding = nn.Linear(input_dim, model_dim)
        self.positional_encoding = nn.Parameter(torch.zeros(1, 8192, model_dim))
        encoder_layers = nn.TransformerEncoderLayer(model_dim, num_heads, dim_feedforward=2048, dropout=dropout_rate)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layers, num_layers)
        self.fc_out = nn.Linear(model_dim, output_dim)

    def forward(self, x):
        x = self.embedding(x.unsqueeze(-1)) + self.positional_encoding[:, :x.size(1), :]
        x = self.transformer_encoder(x)
        x = x.mean(dim=1)  # Global avg pooling
        x = self.fc_out(x)
        return x
```

Figure 39: Defining the Transformer model.

The class inherits from the base class, `nn.Module`, for all Neural Network (NN) modules in PyTorch.

`__init__()` function:

- `nn.Linear(input_dim, model_dim)` is an embedding layer that linearly projects the input from `input_dim` to `model_dim`.
- `nn.Parameter(torch.zeros(1, 8192, model_dim))` creates a positional encoding tensor with shape $(1, 8192, model\_dim)$. This encodes positional information to help the model understand the order of input.
- `nn.TransformerEncoderLayer` defines a transformer encoder layer with:
  - `model_dim`: the dimension of the model.
  - `num_heads`: the number of attention heads.

  - `dim_feedforward=2048`: the dimension of the feedforward network.
  - `dropout=dropout_rate`: the dropout rate.

- `nn.TransformerEncoder` stacks the encoder layers to form the complete transformer encoder.

- `nn.Linear(model_dim, output_dim)` linearly projects the output from `model_dim` to `output_dim`.

`forward()` function:

- `x.unsqueeze(-1)` adds an extra dimension to `x`, making its shape compatible for the embedding layer.

- `self.embedding(x.unsqueeze(-1))` applies the linear transformation to the input.

- `+ self.positional_encoding[:, :x.size(1), :]` adds the positional encoding to the embedded input.

- `self.transformer_encoder(x)` processes the input through the transformer encoder stack.

- `x.mean(dim=1)` performs global average pooling across the sequence dimension, resulting in a tensor of shape $(\text{batch\_size}, \text{model\_dim})$.

- `self.fc_out(x)` linearly transforms the pooled tensor to the desired output dimension.

- The final output tensor is then returned.

```python
def train_and_evaluate(model, train_loader, test_loader, criterion, optimizer, num_epochs):
    train_losses = []
    test_accuracies = []

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        for segments_aug, labels_aug in train_loader:
            optimizer.zero_grad()
            outputs = model(segments_aug)
            loss = criterion(outputs, labels_aug)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        train_loss = running_loss / len(train_loader)
        train_losses.append(train_loss)
        print(f'Epoch {epoch+1}/{num_epochs}, Loss: {train_loss}')

        model.eval()
        correct = 0
        total = 0
        with torch.no_grad():
            for segments_aug, labels_aug in test_loader:
                outputs = model(segments_aug)
                _, predicted = torch.max(outputs.data, 1)
                total += labels_aug.size(0)
                correct += (predicted == labels_aug).sum().item()

        test_accuracy = correct / total
        test_accuracies.append(test_accuracy)

    return train_losses, test_accuracies
```

Figure 40: Defining the function for training and evaluating the Transformer model.

`train_and_evaluate()` function:

- Epoch Loop: iterates over the epochs.
  - `model.train()`: sets the model to training mode.
  - `running_loss` is initialized to 0.0 to accumulate the training loss over all batches in the epoch.
  - Batch Loop: iterates over all batches in the `train_loader`.
    * `optimizer.zero_grad()`: clears the gradients of all optimized parameters.
    * `outputs = model(segments_aug)`: computes the model outputs for the input batch.

    * `loss = criterion(outputs, labels_aug)`: calculates the loss between the predicted outputs and the true labels.
    * `loss.backward()`: computes the gradient of the loss.
    * `optimizer.step()`: updates the model parameters using the computed gradients.
    * `running_loss += loss.item()`: adds the batch loss to the running total loss for the epoch.
  - `train_loss = running_loss / len(train_loader)`: calculates the average training loss for the epoch.
  - `train_losses.append(train_loss)`: appends the average training loss to `train_losses`.
  - `model.eval()`: sets the model to evaluation mode.
  - `with torch.no_grad()`: disables gradient computation, which reduces memory usage and speeds up computations.
  - Batch Loop: iterates over all batches in the `test_loader`.
    * `outputs = model(segments_aug)`: computes the model outputs for the input batch.
    * `_, predicted = torch.max(outputs.data, 1)`: finds the one with the highest predicted score for each sample.
    * `total += labels_aug.size(0)` and `correct += (predicted == labels_aug).sum().item()`: updates the total number of samples and the number of correct predictions.
- The function returns two lists: `train_losses`, containing the average training loss for each epoch, and `test_accuracies`, containing the test accuracy for each epoch.

```python
# Def model
model = TransformerModel(input_dim, model_dim, num_heads, num_layers, dropout_rate, output_dim)

# Train model
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=lr)
train_losses, test_accuracies = train_and_evaluate(model, train_loader, test_loader, criterion, optimizer, num_epochs)
```

Figure 41: Building and training the Transformer model.

5.4 DBN

A DBN is trained for binary classification, capturing hierarchical representations in the data.

```python
# Def DBN
class DBN(nn.Module):
    def __init__(self):
        super(DBN, self).__init__()
        self.layer1 = nn.Linear(X_train_aug.shape[1], 256)
        self.layer2 = nn.Linear(256, 128)
        self.layer3 = nn.Linear(128, 64)
        self.output = nn.Linear(64, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.layer1(x))
        x = self.sigmoid(self.layer2(x))
        x = self.sigmoid(self.layer3(x))
        x = self.sigmoid(self.output(x))
        return x

model = DBN()
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Figure 42: Defining the DBN model.

The class inherits from the base class, `nn.Module`, for all NN modules in PyTorch.

`__init__()` function:

- `self.layer1` takes the input data and outputs 256 features.
- `self.layer2` takes the 256 features from `layer1` and outputs 128 features.
- `self.layer3` takes the 128 features from `layer2` and outputs 64 features.
- `self.output` takes the 64 features from `layer3` and outputs a single feature for binary classification or regression.
- Sigmoid activation is used.

`forward()` function:

- It defines the forward pass of the network, which is the way input data flows through the network shown in the constructor.

- The final output `x` is returned. It will be in the range of (0, 1), which is fit for binary classification tasks.

```python
# Train model
train_losses = []
train_accuracies = []
val_losses = []
val_accuracies = []

epochs = 100
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(X_train_aug)
    loss = criterion(outputs, y_train_aug)
    loss.backward()
    optimizer.step()

    # Calc training accuracy
    predicted = (outputs >= 0.5).float()
    accuracy = (predicted.eq(y_train_aug).sum() / float(y_train_aug.shape[0])).item()

    train_losses.append(loss.item())
    train_accuracies.append(accuracy)

    # Evalute model
    with torch.no_grad():
        val_outputs = model(X_test)
        val_loss = criterion(val_outputs, y_test)
        val_predicted = (val_outputs >= 0.5).float()
        val_accuracy = (val_predicted.eq(y_test).sum() / float(y_test.shape[0])).item()

        val_losses.append(val_loss.item())
        val_accuracies.append(val_accuracy)

    print(f'Epoch [{epoch+1}/{epochs}]\n  Loss: {loss.item():.4f}, Accuracy: {accuracy:.4f}\n  Val Loss: {val_loss.item():.4f}, Val Accuracy: {val_accuracy:.4f}')
```

Figure 43: Training and evaluating the DBN model.

Epoch Loop: iterates over the epochs.

- `model.train()`: sets the model to training.
- `optimizer.zero_grad()`: clears the gradients of all optimized parameters.
- `outputs = model(X_train_aug)`: processes the input data `X_train_aug` and produces outputs.
- `loss = criterion(outputs, y_train_aug)`: calculates the difference between the outputs and the true labels `y_train_aug`.
- `loss.backward()`: performs backpropagation to compute the gradients of the loss respective to the parameters.
- `optimizer.step()`: updates the parameters using the computed gradients.
- `predicted = (outputs >= 0.5).float()`: converts the outputs to binary predictions with a threshold of 0.5.

- `accuracy = (predicted.eq(y_train_aug).sum() / float(y_train_aug.shape[0])).item()`: compares the predicted labels to the true labels and calculates the accuracy.
- `with torch.no_grad()`: disables gradient computation, which reduces the memory used and speeds up computations.
- `val_outputs = model(X_test)`: processes the test data `X_test`.
- `val_loss = criterion(val_outputs, y_test)`: calculates the difference between the outputs and the true labels `y_test`.
- `val_predicted = (val_outputs >= 0.5).float()`: converts the outputs to binary predictions.
- `val_accuracy = (val_predicted.eq(y_test).sum() / float(y_test.shape[0])).item()`: calculates the accuracy of the predictions on the test data.

5.5 GNN

A GNN is trained for classification, using the graph structure of the data to capture complex relationships.

```python
# Def GNN
class GNN(torch.nn.Module):
    def __init__(self, in_channels):
        super(GNN, self).__init__()
        self.conv1 = GCNConv(in_channels=in_channels, out_channels=16)
        self.conv2 = GCNConv(in_channels=16, out_channels=32)
        self.fc = torch.nn.Linear(32, 2)  # Binary classification


    def forward(self, data):
        x, edge_index, batch = data.x, data.edge_index, data.batch
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.conv2(x, edge_index)
        x = F.relu(x)
        x = global_mean_pool(x, batch)
        x = self.fc(x)
        return F.log_softmax(x, dim=1)

in_channels = graph_data[0].x.shape[1]
model = GNN(in_channels=in_channels)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = torch.nn.CrossEntropyLoss()
```

Figure 44: Defining the GNN model.

The class inherits from the base class, `nn.Module`, for all NN modules in PyTorch.

`__init__()` function:

- `self.conv1 = GCNConv(in_channels=in_channels, out_channels=16)`: initializes the first graph convolutional layer with input data and 16 output features.
- `self.conv2 = GCNConv(in_channels=16, out_channels=32)`: initializes the second graph convolutional layer with 16 input features from the first layer and 32 output features.
- `self.fc = torch.nn.Linear(32, 2)`: initializes a fully connected layer that takes 32 input features from the second layer and outputs 2 features used for binary classification.

`forward() function`:

- It defines the forward pass of the network, which is the way input data flows through the network shown in the constructor.

- `x = F.relu(x)`: applies the ReLU activation.
- `x = global_mean_pool(x, batch)`: applies global mean pooling to obtain a graph-level representation.
- `x = self.fc(x)`: applies the fully connected layer to the graph-level representation.
- `return F.log_softmax(x, dim=1)`: applies the log softmax function, converting the raw scores into log-probabilities for classification tasks.

```python
# Training history
train_losses = []
train_accuracies = []

# Training loop
def train():
    model.train()
    epoch_loss = 0
    correct = 0
    for data in data_loader:
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, data.y)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
        pred = output.argmax(dim=1)
        correct += (pred == data.y).sum().item()
    train_losses.append(epoch_loss / len(data_loader))
    print(f"Loss: {epoch_loss / len(data_loader)}")
    train_accuracies.append(correct / len(graph_data))
    print(f"Accuracy: {correct / len(graph_data)}")

# Train & evalute GNN
epochs = 10
for epoch in range(epochs):
    print(f"Epoch {epoch+1}/{epochs}")
    train()
```

Figure 45: Training and evaluating the GNN model.

`train()` function:

- `model.train()`: sets the model to training.
- Batch Loop: Iterates over all batches in the `data_loader`.

- `optimizer.zero_grad()`: clears the gradients of all optimized parameters.
- `output = model(data)`: passes the input data to get predictions.
- `loss = criterion(output, data.y)`: calculates the loss between the output and the true labels.
- `loss.backward()`: compute the gradient of the loss respective to the parameters.
- `optimizer.step()`: update the parameters with the computed gradients.
- `.item()` converts the tensor to a number.
- `pred = output.argmax(dim=1)`: obtain the prediction with the index of the highest log probability.

Epoch Loop: applies `train()` function over epochs.

5.6 GAN

5.6.1 Hyperparameters

```
latent_dim = 2
epochs = 5
batch_size = 64
num_gw_data_to_generate = 10
```

Figure 46: The hyperparameters for implementing GAN.

- `latent_dim`: dimensionality of the latent space (input vector)).
- `num_gw_data_to_generate`: number of synthetic gravitational wave data samples to generate after training.

5.6.2 Define Generator

The `build_generator` function creates the generator model to synthesize GW data.

```python
def build_generator(latent_dim):
    model = Sequential()
    model.add(Dense(256 * 1024, input_dim=latent_dim))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Reshape((1024, 256)))
    model.add(UpSampling1D())
    model.add(Conv1D(128, kernel_size=3, padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(UpSampling1D())
    model.add(Conv1D(64, kernel_size=3, padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(UpSampling1D())
    model.add(Conv1D(1, kernel_size=3, padding='same', activation='tanh'))
    return model
```

Figure 47: The construction of the build generator.

- `Dense Layer`: initial dense layer with `latent_dim` input.
- `LeakyReLU`: LeakyReLU activation function.
- `BatchNormalization`: normalizes the output.
- `Reshape`: reshapes the output into a suitable shape for `Conv1D` layers.
- `UpSampling1D`: upsamples the input.
- `Conv1D Layers`: convolutional layers to extract features.
- `Activation`: `tanh` activation to output values between -1 and 1.

5.6.3 Define Discriminator

The `build_discriminator` function creates the discriminator model to distinguish real versus generated data.

```
def build_discriminator(input_shape):
    model = Sequential()
    model.add(Conv1D(64, kernel_size=3, strides=2, input_shape=input_shape, padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Conv1D(128, kernel_size=3, strides=2, padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))
    return model
```

Figure 48: The construction of the build generator.

- `Conv1D Layers`: convolutional layers to extract features.
- `LeakyReLU`: LeakyReLU activation function.
- `Flatten`: Flattens the 3D tensor into 1D.
- `Dense Layer`: final dense layer to output a single probability (of it being real and not generated data) with sigmoid activation.

5.6.4 Define GAN

The `build_gan` function combines the generator and discriminator into a GAN model.

```
def build_gan(generator, discriminator):
    discriminator.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    discriminator.trainable = False
    gan_input = tf.keras.Input(shape=(latent_dim,))
    generated_data = generator(gan_input)
    gan_output = discriminator(generated_data)
    gan = tf.keras.Model(gan_input, gan_output)
    gan.compile(loss='binary_crossentropy', optimizer='adam')
    return gan
```

Figure 49: The construction of the GAN.

- `Compile Discriminator`: compile the discriminator.
- `Freeze Discriminator`: ensure only the generator is trained.
- `GAN Input`: create input layer for the GAN model.
- `Generated Data`: pass input through the generator to get synthetic data.
- `GAN Output`: pass generated data through the discriminator to get the probability (of it being real and not generated data).
- `Compile GAN`: compile the GAN model.

5.6.5 Train GAN

The `train_gan` function trains the GAN by alternating between training the discriminator and the generator. The steps are as follows:

```
def train_gan(generator, discriminator, gan, data, epochs, batch_size, latent_dim):
    half_batch = int(batch_size / 2)
    d_losses = []
    g_losses = []

    for epoch in range(1, epochs+1):
        # Train Discriminator
        idx = np.random.randint(0, data.shape[0], half_batch)
        real_data = data[idx]

        noise = np.random.normal(0, 1, (half_batch, latent_dim))
        generated_data = generator.predict(noise)

        d_loss_real = discriminator.train_on_batch(real_data, np.ones((half_batch, 1)))
        d_loss_fake = discriminator.train_on_batch(generated_data, np.zeros((half_batch, 1)))
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # Train Generator
        noise = np.random.normal(0, 1, (batch_size, latent_dim))
        valid_y = np.array([1] * batch_size)
        g_loss = gan.train_on_batch(noise, valid_y)

        # Save losses
        d_losses.append(d_loss[0])
        g_losses.append(g_loss)

        print(f"Epoch {epoch}\n  D loss: {d_loss[0]}\n  D accuracy: {d_loss[1]}\n  G loss: {g_loss}")
    return d_losses, g_losses
```

Figure 50: Visualization of the GAN training loop.

Training Loop:

- `Train Discriminator`:
  - Sample real data.
  - Generate synthetic data.
  - Train on real data (labeled 1) and synthetic data (labeled 0).
  - Compute the discriminator loss.
- `Train Generator`:
  - Generate random noise.
  - Create an array with every element labeled 1 for the noise.

- Train on the random noise and array.
- Compute the generator loss.

5.7 WaveNet

5.7.1 Define Causal Convolutional Layer

The causal convolutional layers are used to maintain causality in time series data.

```
class CausalConv1D(layers.Layer):
    def __init__(self, filters, kernel_size, dilation_rate, **kwargs):
        super(CausalConv1D, self).__init__(**kwargs)
        self.conv = layers.Conv1D(filters, kernel_size, padding='causal', dilation_rate=dilation_rate)

    def call(self, x):
        return self.conv(x)
```

Figure 51: The causal Conv1D class.

CausalConv1D Class:

- __init__ function:
  - Inherits from `layers.Layer`.
  - Creates a `Conv1D` layer.
- Call function:
  - Defines the forward pass by returning the convolutional layer that's applied to the input tensor.

5.7.2 Define Residual Block

The residual block is added to build complex feature representations while maintaining gradient flow through skip connections.

```python
class ResidualBlock(layers.Layer):
    def __init__(self, filters, kernel_size, dilation_rate, **kwargs):
        super(ResidualBlock, self).__init__(**kwargs)
        self.causal_conv = CausalConv1D(filters, kernel_size, dilation_rate)
        self.dense_tanh = layers.Dense(filters, activation='tanh')
        self.dense_sigmoid = layers.Dense(filters, activation='sigmoid')
        self.skip_conv = layers.Conv1D(filters, 1)
        self.residual_conv = layers.Conv1D(filters, 1)

    def call(self, x):
        out = self.causal_conv(x)
        tanh_out = self.dense_tanh(out)
        sigmoid_out = self.dense_sigmoid(out)
        gated_activation = tanh_out * sigmoid_out
        skip_out = self.skip_conv(gated_activation)
        residual_out = self.residual_conv(gated_activation)
        return skip_out, x + residual_out
```

Figure 52: The residual block class.

ResidualBlock Class:

- __init__ function:
  - Inherits from `layers.Layer`.
  - Creates a `CausalConv1D` layer.
  - Creates two `Dense` layers with `tanh` and `sigmoid` activations, respectively.
  - Creates two `Conv1D` layers for skip and residual connections.
- Call function:
  - Defines the forward pass:
    * Applying the `CausalConv1D` layer to the input.
    * Applying the `Dense` layers with `tanh` and `sigmoid` activations to the output of the previous layer.
    * Multiplying the outputs of the `tanh` and `sigmoid` layers to create a gated activation.
    * Applying the `skip_conv` layer to the gated activation for the skip connection.
    * Applying the `residual_conv` layer to the gated activation and adding it to the input to create the residual output.
  - Returning the skip output and the residual output.

5.7.3 Define and Train WaveNet

The build_wavenet function creates a WaveNet model for sequential data generation.

```
def build_wavenet(input_shape, filters, kernel_size, dilation_rates):
    inputs = tf.keras.Input(shape=input_shape)
    x = inputs
    skip_connections = []

    for dilation_rate in dilation_rates:
        skip_out, x = ResidualBlock(filters, kernel_size, dilation_rate)(x)
        skip_connections.append(skip_out)

    x = layers.Add()(skip_connections)
    x = layers.Activation('relu')(x)
    x = layers.Conv1D(filters, 1, activation='relu')(x)
    x = layers.Conv1D(1, 1)(x)

    return tf.keras.Model(inputs, x)
```

Figure 53: The construction of the WaveNet.

- `Inputs`: input layer with specified shape.
- `Residual Blocks`: apply multiple residual blocks with different dilation rates.
- `Skip Connections`: collect and sum connections.
- `Activations and Convolutions`: layers to produce output.

We then train the WaveNet on augmented data and validate it on test data.

```
history = model.fit(X_train_aug, y_train_aug, epochs=10, batch_size=128, validation_data=(X_test, y_test))
```

Figure 54: Training and saving its history for WaveNet

5.8 Traditional ML Models

Traditional ML models, including SVM, RF, and GMM, offer versatile solutions for various predictive tasks. SVM is a powerful supervised learning algorithm useful for classification purposes, which aims to best separate classes in a dataset based on the information present. RF, an ensemble method, constructs multiple decision trees and combines their predictions to improve accuracy and reduce overfitting, making them robust for classification. GMM is an unsupervised learning algorithm used for clustering, organizing data into a mixture of several Gaussian distributions to identify the underlying patterns in complex datasets. These traditional ML models are still widely used due to their effectiveness in many applications, though their usage in GW astronomy is less commonly associated since certain DL models, such as CNNs and RNNs, already display promising results in binary classification.

5.8.1 SVM

An SVM model with an RBF kernel is trained with the training data and evaluated with the validation data. The confusion matrix and classification report provide insights into the model's performance.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	410
accuracy			1.00	410
macro avg	1.00	1.00	1.00	410
weighted avg	1.00	1.00	1.00	410

Figure 55: The confusion matrix and classification report for SVM.

5.8.2 RF

A RF model is trained and evaluated similarly. The confusion matrix and classification report are also applied to examine its performance.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	410
accuracy			1.00	410
macro avg	1.00	1.00	1.00	410
weighted avg	1.00	1.00	1.00	410

Figure 56: The confusion matrix and classification report for RF.

5.8.3 GMM

A GMM is trained on the original segment data due to its unsupervised nature. The log-likelihood of the data is computed and used to detect outliers, defined as the bottom 0.01% of the log-likelihood value, and these outliers represent a higher likelihood of a GW event present at the corresponding time.

Figure 57: Number of outliers detected with GMM considering the bottom 0.01% of the data as outliers.

6 Model Performance Visualization

6.1 1D CNN

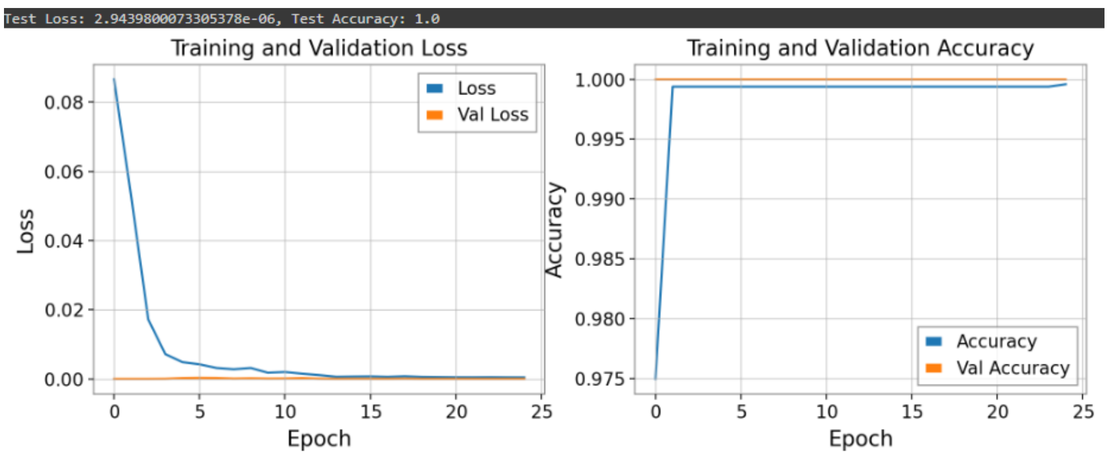

Figure 58: These plots show the training history of the 1D CNN, including the test loss and accuracy evaluation.

6.2 2D CNN

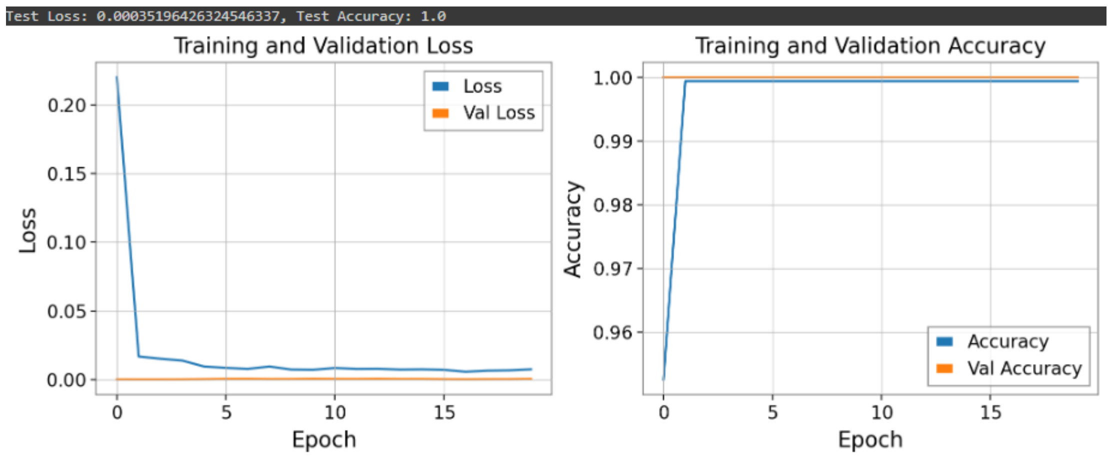

Figure 59: These plots show the training history of the 2D CNN, including the test loss and accuracy evaluation.

6.3 LSTM

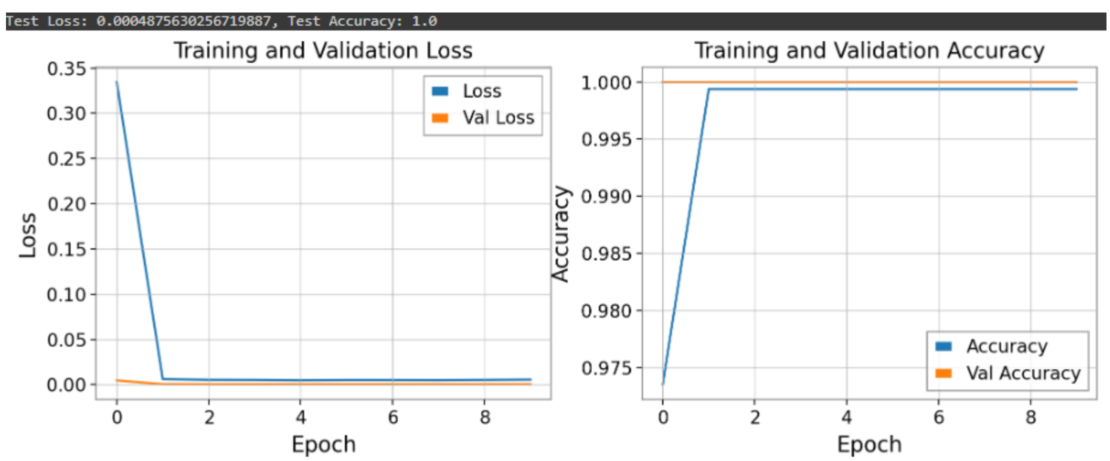

Figure 60: These plots show the training history of the LSTM, including the test loss and accuracy evaluation.

6.4 GRU

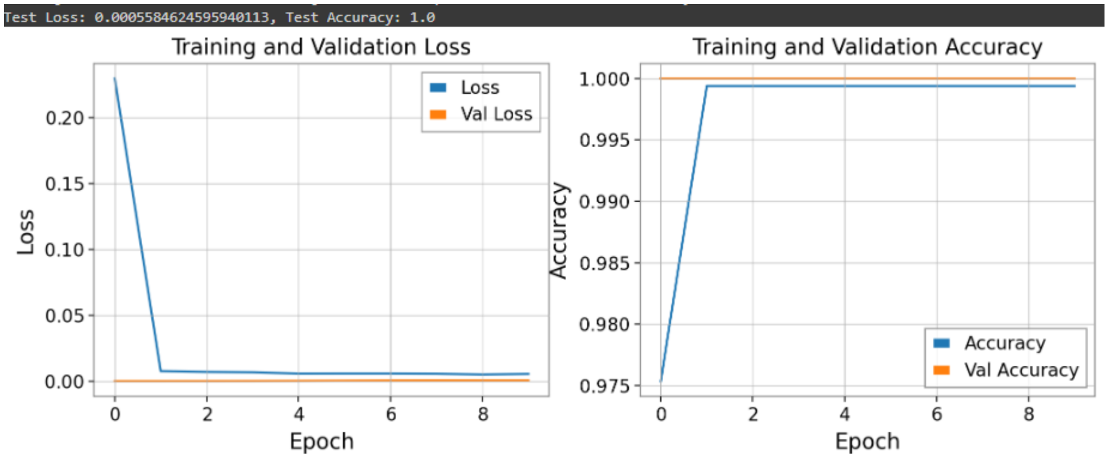

Figure 61: These plots show the training history of the GRU, including the test loss and accuracy evaluation.

6.5 1D CNN Autoencoder

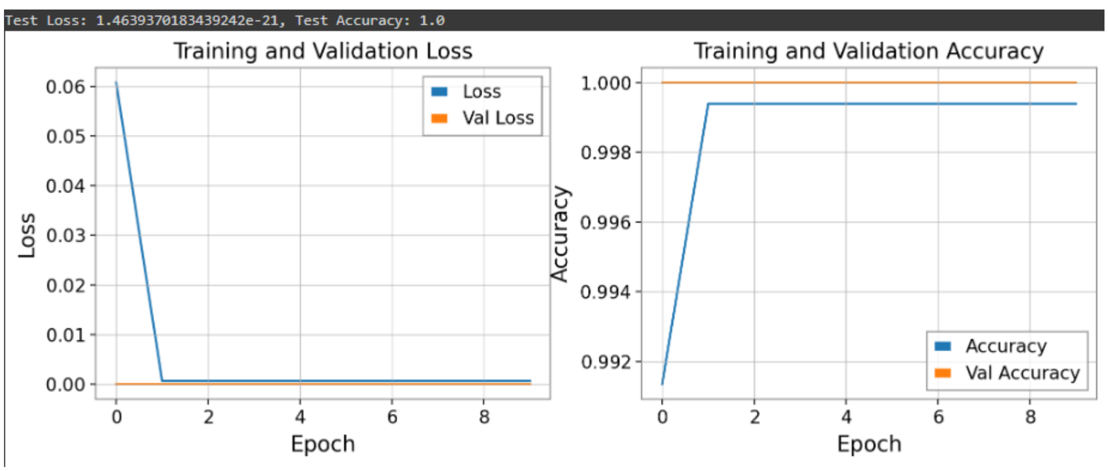

Figure 62: These plots show the training history of the 1D CNN autoencoder, including the test loss and accuracy evaluation.

6.6 2D CNN Autoencoder

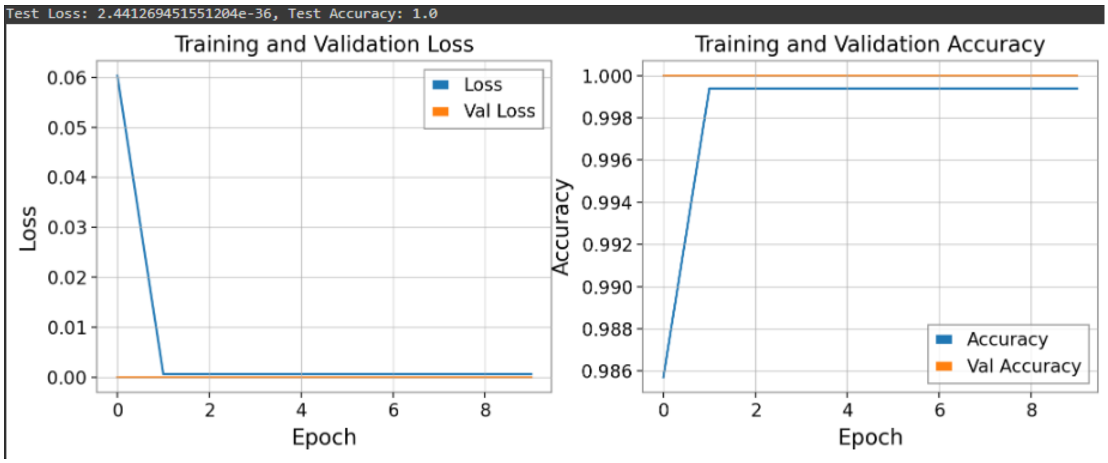

Figure 63: These plots show the training history of the 2D CNN autoencoder, including the test loss and accuracy evaluation.

6.7 LSTM Autoencoder

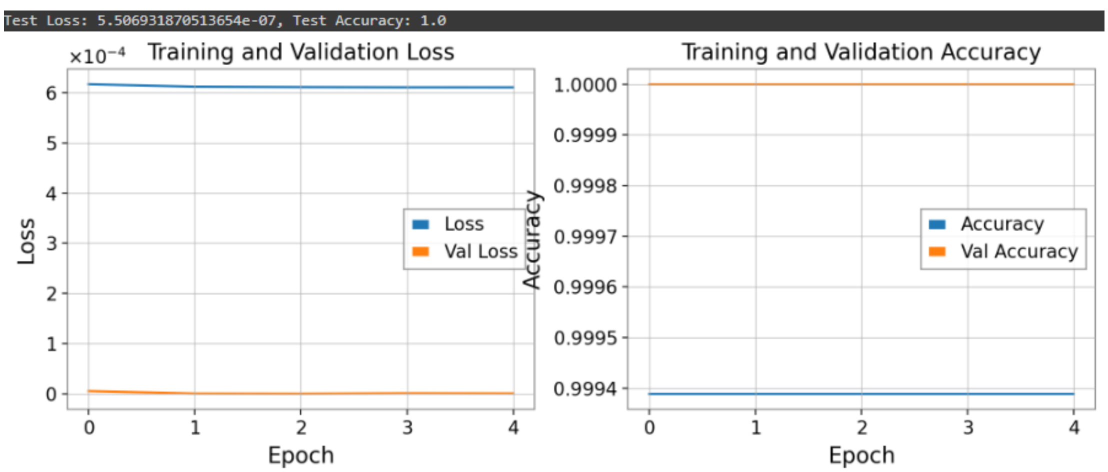

Figure 64: These plots show the training history of the LSTM autoencoder, including the test loss and accuracy evaluation.

6.8 GRU Autoencoder

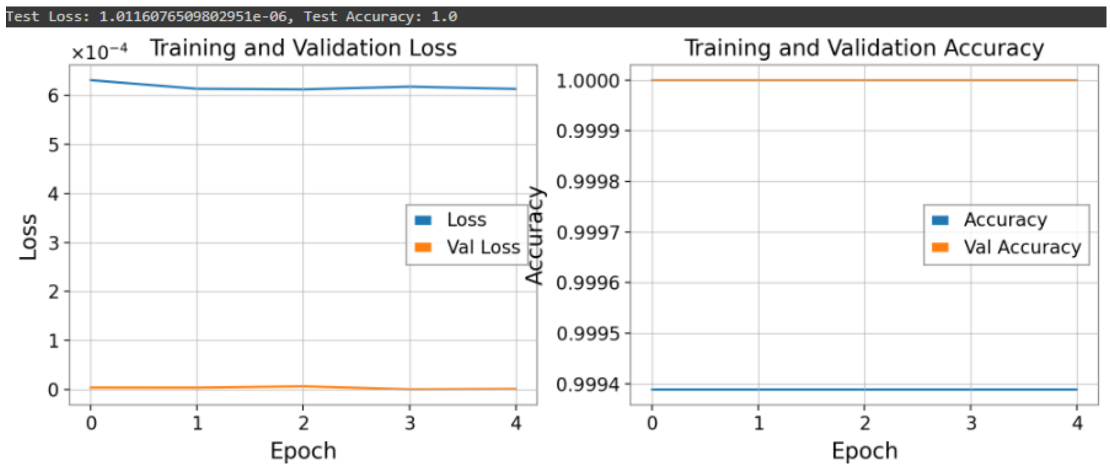

Figure 65: These plots show the training history of the GRU autoencoder, including the test loss and accuracy evaluation.

6.9 Transformer

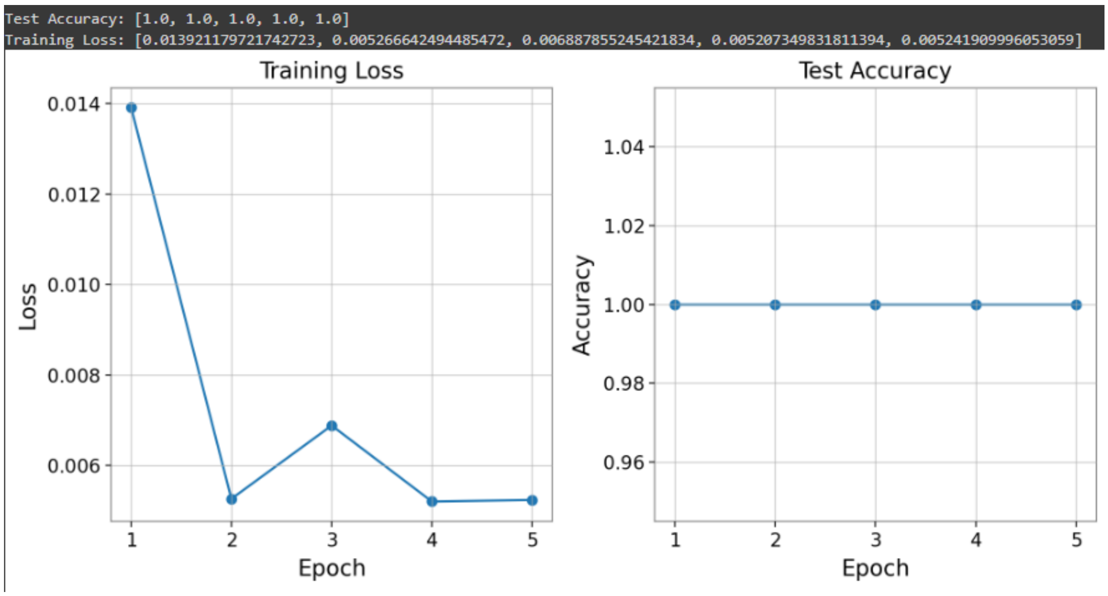

Figure 66: These plots show the training history of the Transformer model, including the loss and accuracy evaluation.

6.10 DBN

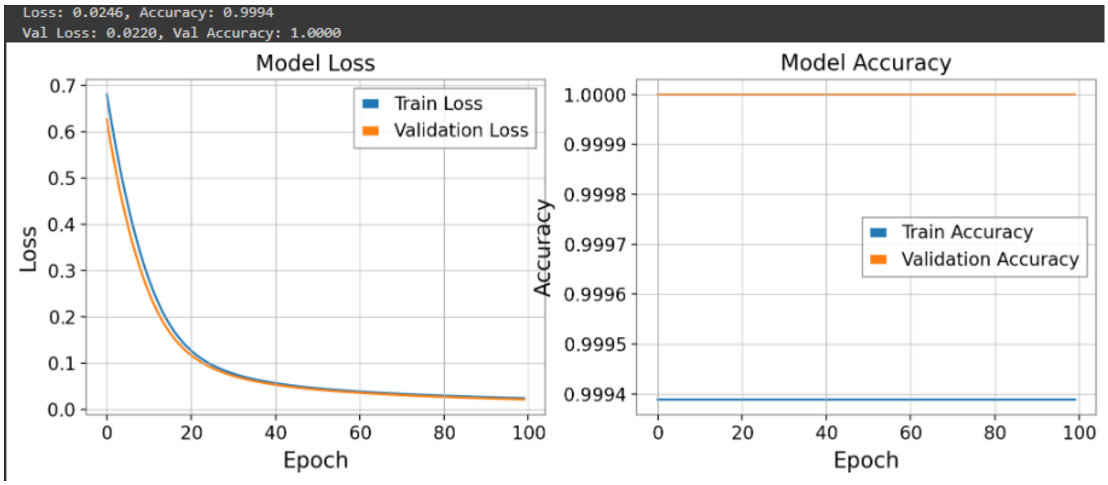

Figure 67: These plots show the training history of the DBN model, including the loss and accuracy evaluation.

6.11 GNN

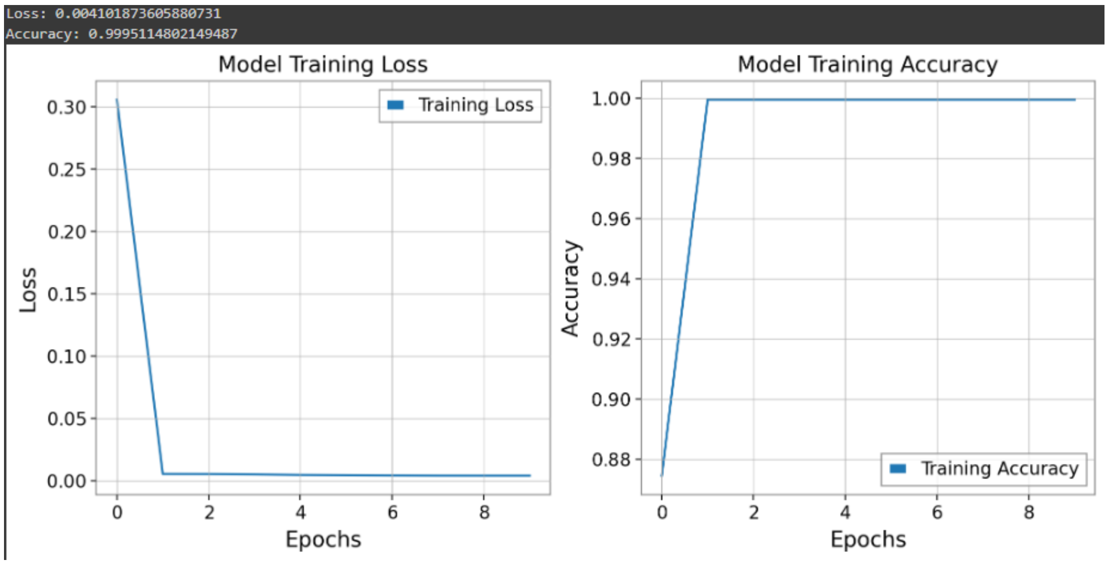

Figure 68: These plots show the training history of the GNN model, including the loss and accuracy evaluation.

6.12 GAN

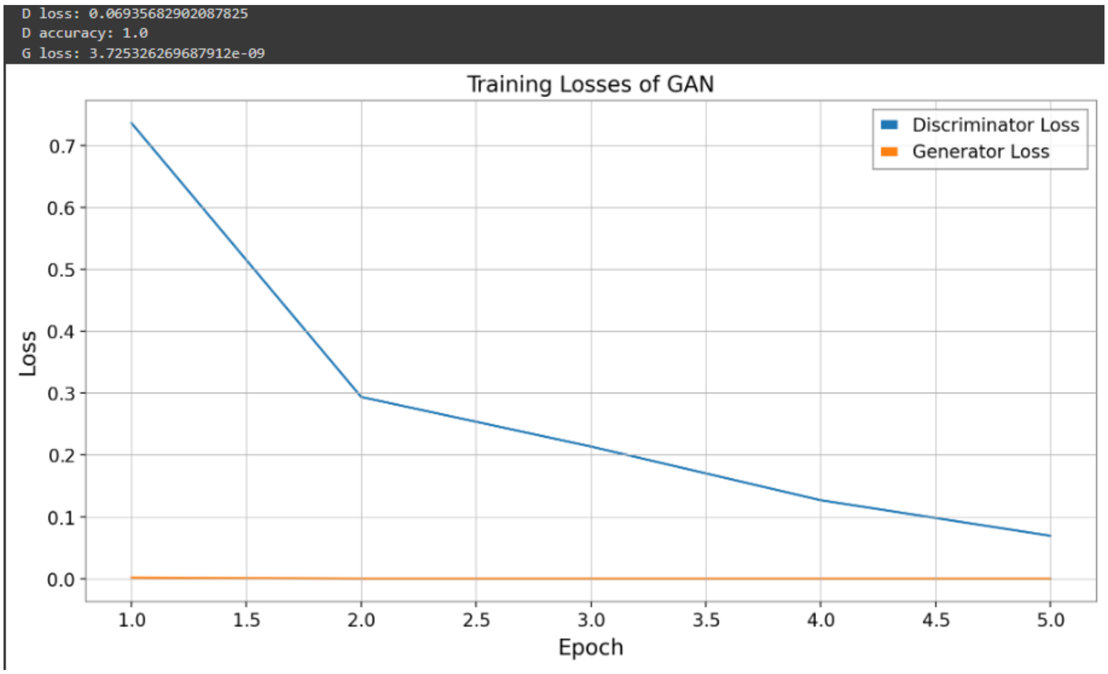

Figure 69: Visualization of the discriminator and generator losses over epochs (G = Generator, D = Discriminator).

6.13 WaveNet

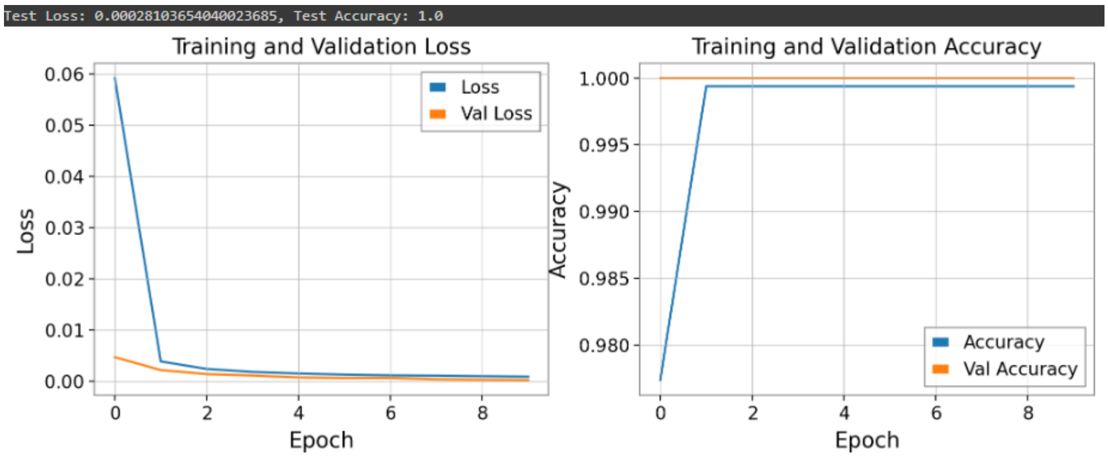

Figure 70: Visualization of the loss and accuracy over epochs for WaveNet model.

6.14 SVM (ROC Curve)

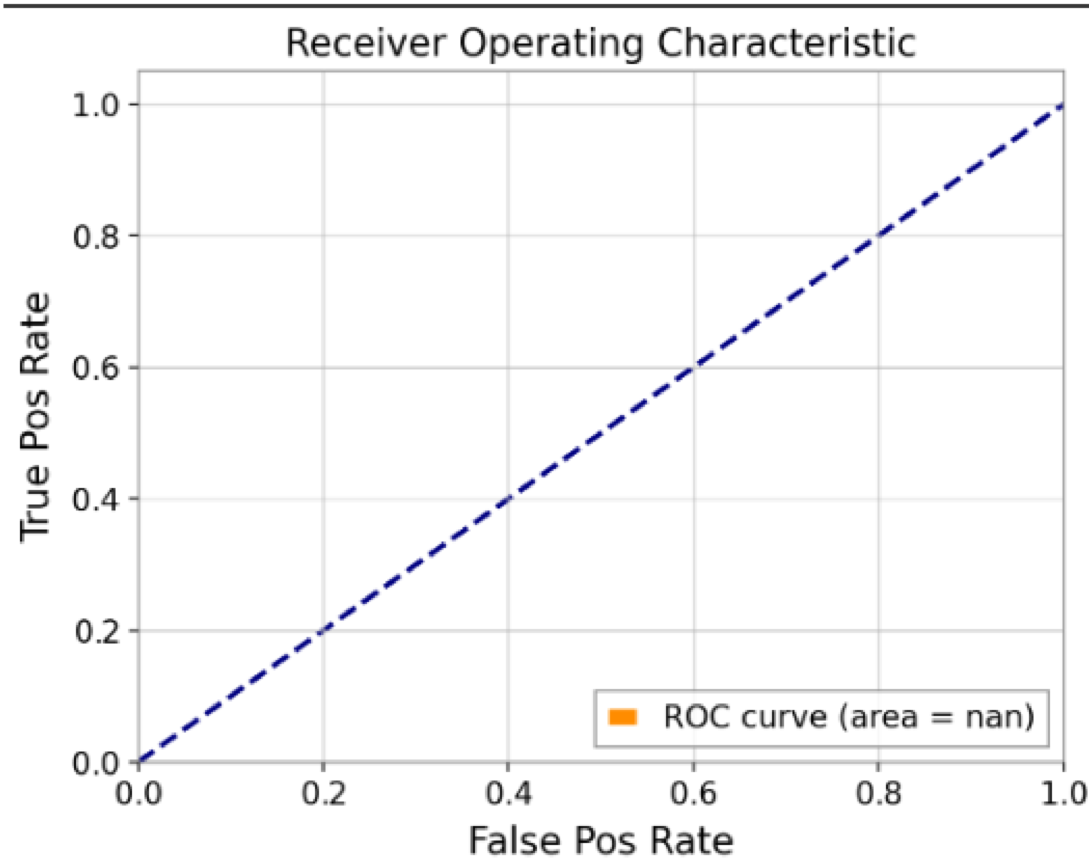

Figure 71: The ROC curve for SVM.

6.15 RF (ROC Curve)

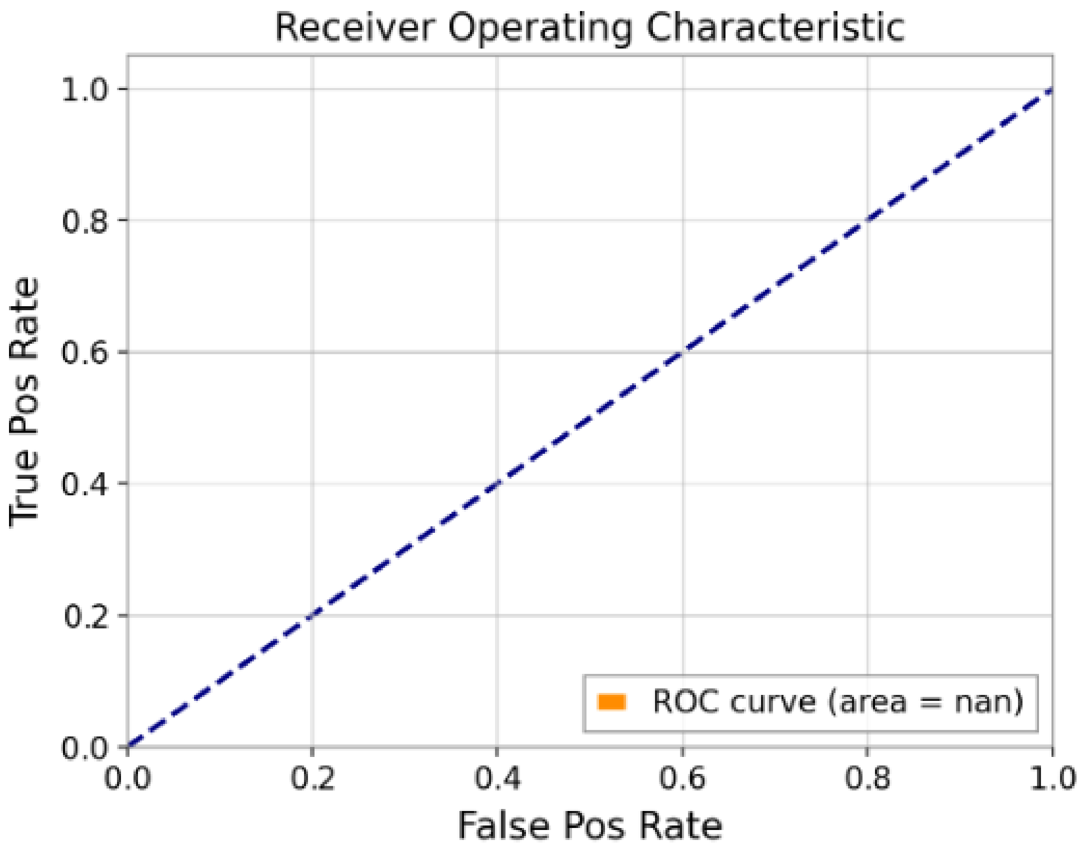

Figure 72: The ROC curve for RF.

6.16 GMM (Clustering)

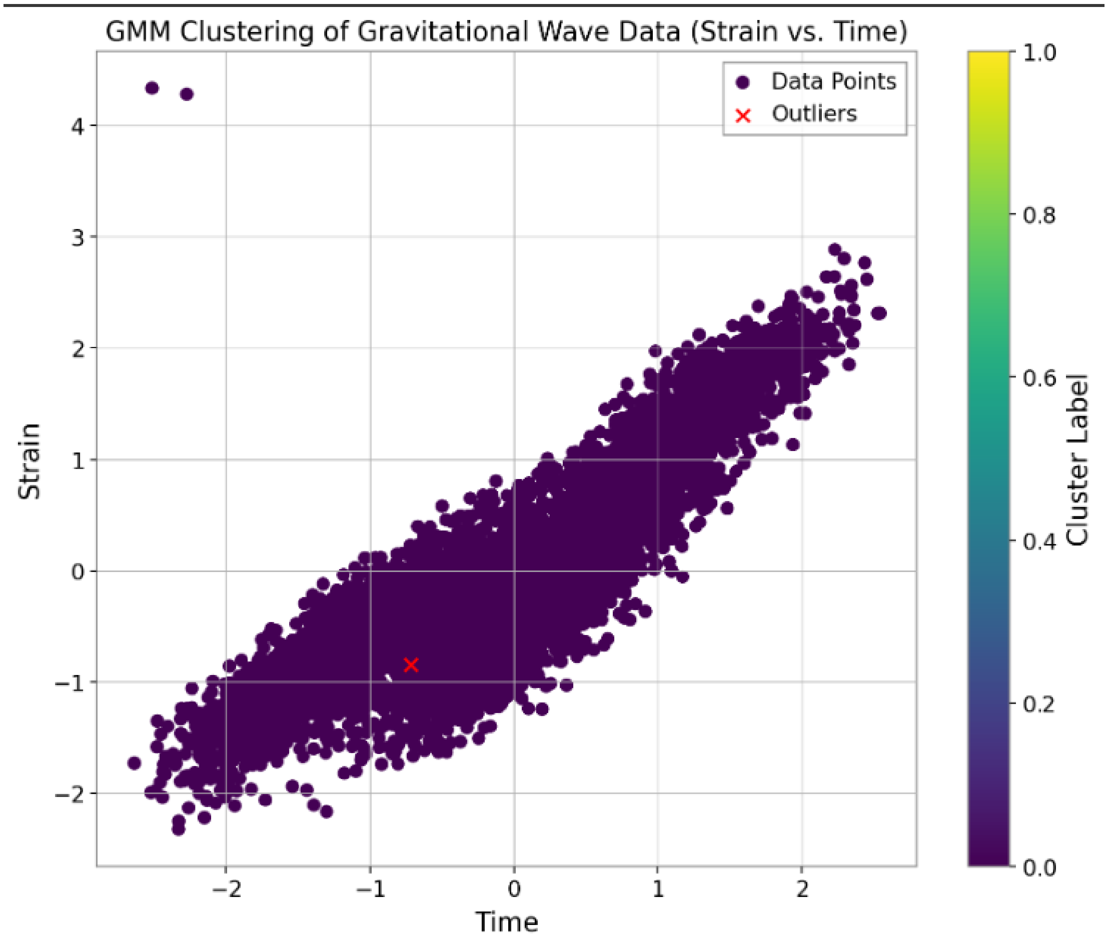

Figure 73: The clustering results from GMM, highlighting detected outliers and clusters in the data.

7 Conclusion

The advancements in GW astronomy have profoundly expanded our understanding of the universe, enabling the exploration of some of its most energetic and enigmatic phenomena. The integration of ML techniques into GW data analysis marks a significant milestone, allowing for the detection, classification, and analysis of GW signals with unprecedented accuracy and precision. ML models such as CNNs, RNNs, and many others have become indispensable in extracting faint GW signals from highly noisy data environments.

Specifically, synthetic data generation through models like GANs and WaveNet plays a vital role in augmenting training datasets, especially in situations where real data is scarce or specific events, such as rare mergers, are underrepresented. The synthetic data, designed to mimic the characteristics of real GW signals, can replace the data generated with the traditional augmentation technique and improve the training of ML models by introducing controlled variability, enabling these models to generalize better to diverse and complex scenarios. This not only enhances the models' ability to identify and classify GW events but also improves their sensitivity to subtle patterns that might otherwise be overlooked.

As the field advances, the connection between increasingly sophisticated ML algorithms and the rapidly expanding global network of GW detectors will undoubtedly lead to more profound discoveries. Continued innovations in event classification and synthetic data augmentation techniques will be critical to refining our models, making them more resilient to noise and better equipped to handle rare events. These developments promise to push the boundaries of our knowledge of high-energy astrophysics, the dynamics of compact objects, and the widening field of gravity.

Looking ahead, the fusion of ML with GW astronomy will remain pivotal in deepening our insights into fundamental physics, cosmology, and the evolution of black holes and neutron stars. With ongoing improvements in both detection technology and analytical methodologies, the future of GW astronomy is bright, offering the potential for even deeper understanding and discoveries about the cosmos.

References

[1] Abbott, B.P., et al. "Observation of Gravitational Waves from a Binary Black Hole Merger." *Physical Review Letters*, vol. 116, 061102, 2016.

[2] Abbott, B.P., et al. "GW170817: Observation of Gravitational Waves from a Binary Neutron Star Inspiral." *Physical Review Letters*, vol. 119, 161101, 2017.

[3] Abbott, R., et al. "GWTC-2: Compact Binary Coalescences Observed by LIGO and Virgo during the First Half of the Third Observing Run." *Physical Review X*, vol. 11, 021053, 2021.

[4] Abbott, B.P., et al. "Tests of General Relativity with GW150914." *Physical Review Letters*, vol. 116, 221101, 2016.

[5] Abbott, B.P., et al. "Multi-Messenger Observations of a Binary Neutron Star Merger." *Astrophysical Journal Letters*, vol. 848, L12, 2017.

[6] Abbott, B.P., et al. "GWTC-1: A Gravitational-Wave Transient Catalog of Compact Binary Mergers Observed by LIGO and Virgo during the First and Second Observing Runs." *Physical Review X*, vol. 9, 031040, 2019.

[7] Abbott, B.P., et al. "Binary Black Hole Mergers in the First Advanced LIGO Observing Run." *Physical Review X*, vol. 6, 041015, 2016.

[8] Abbott, B.P., et al. "Properties of the Binary Black Hole Merger GW150914." *Physical Review Letters*, vol. 116, 241102, 2016.

[9] Abbott, B.P., et al. "GW170104: Observation of a 50-Solar-Mass Binary Black Hole Coalescence at Redshift 0.2." *Physical Review Letters*, vol. 118, 221101, 2017.

[10] Abbott, B.P., et al. "Astrophysical Implications of the Binary Black-Hole Merger GW150914." *Astrophysical Journal Letters*, vol. 818, L22, 2016.

[11] Zheng, Y., et al. "Angular Power Spectrum of Gravitational-Wave Transient Sources as a Probe of the Large-Scale Structure." *Physical Review Letters*, vol. 131, 171403, 2023.

[12] Abbott, B.P., et al. "Upper Limits on the Stochastic Gravitational-Wave Background from Advanced LIGO's First Observing Run." *Physical Review Letters*, vol. 118, 121102, 2017.

[13] Abbott, B.P., et al. "Localization and Broadband Follow-up of the Gravitational-Wave Transient GW150914." *Astrophysical Journal Letters*, vol. 826, L13, 2016.

[14] Abbott, B.P., et al. "GW170608: Observation of a 19-solar-mass Binary Black Hole Coalescence." *Astrophysical Journal Letters*, vol. 851, L35, 2017.

[15] Abbott, B.P., et al. "GW170814: A Three-Detector Observation of Gravitational Waves from a Binary Black Hole Coalescence." *Physical Review Letters*, vol. 119, 141101, 2017.

[16] Abbott, B.P., et al. "GW170104: Observation of a 50-Solar-Mass Binary Black Hole Coalescence at Redshift 0.2." *Physical Review Letters*, vol. 118, 221101, 2017.

[17] Abbott, B.P., et al. "GW151226: Observation of Gravitational Waves from a 22-Solar-Mass Binary Black Hole Coalescence." *Physical Review Letters*, vol. 116, 241103, 2016.

[18] Abbott, B.P., et al. "Binary Black Hole Mergers in the First Advanced LIGO Observing Run." *Physical Review X*, vol. 6, 041015, 2016.

[19] LIGO Scientific Collaboration, et al. "Advanced LIGO." *Classical and Quantum Gravity*, vol. 32, 074001, 2015.

[20] Acernese, F., et al. "Advanced Virgo: A Second-Generation Interferometric Gravitational Wave Detector." *Classical and Quantum Gravity*, vol. 32, 024001, 2015.

[21] Aasi, J., et al. "Characterization of the LIGO Detectors during Their Sixth Science Run." *Classical and Quantum Gravity*, vol. 32, 115012, 2015.

[22] The Virgo Collaboration, et al. "Observation of Gravitational Waves from a Binary Black Hole Merger by the Advanced LIGO and Virgo Collaborations." *Physical Review Letters*, vol. 119, 141101, 2017.

[23] Abadie, J., et al. "Topologies for Future Gravitational Wave Detectors." *Classical and Quantum Gravity*, vol. 27, 173001, 2010.

[24] Ajith, P., et al. "A Template Bank for Gravitational Waveforms from Coalescing Binary Black Holes: I. Non-Spinning Binaries." *Physical Review D*, vol. 77, 104017, 2008.

[25] Kalogera, V., et al. "The Gravitational-Wave Breakthrough: Birth of the New Astronomy." *Physics Today*, vol. 70, no. 8, 2017.

[26] Sathyaprakash, B.S., et al. "Scientific Objectives of Einstein Telescope." *Classical and Quantum Gravity*, vol. 29, 124013, 2012.

[27] Punturo, M., et al. "The Einstein Telescope: A Third-Generation Gravitational Wave Observatory." *Classical and Quantum Gravity*, vol. 27, 194002, 2010.

[28] Sesana, A., et al. "Multi-band Gravitational-wave Astronomy: Science with Joint Space- and Ground-based Observations of Black Hole Binaries." *Astrophysical Journal*, vol. 817, 2016.

[29] Cutler, C., and Thorne, K.S. "An Overview of Gravitational-Wave Sources." *Proceedings of the GR16 Conference on General Relativity and Gravitation*, vol. 10, 2001.

[30] Cornish, N.J., et al. "Detecting a Stochastic Gravitational Wave Background in the Presence of a Galactic Foreground." *Physical Review D*, vol. 84, 062003, 2011.

[31] Maggiore, M. "Gravitational Waves: Volume 1: Theory and Experiments." *Oxford University Press*, 2007.

[32] Moore, C.J., et al. "Gravitational-wave Sensitivity Curve Plotter." *Classical and Quantum Gravity*, vol. 32, 015014, 2014.

[33] Blanchet, L. "Gravitational Radiation from Post-Newtonian Sources and Inspiralling Compact Binaries." *Living Reviews in Relativity*, vol. 17, 2014.

[34] Thorne, K.S. "Gravitational Radiation." *Proceedings of the GR7 Conference on General Relativity and Gravitation*, vol. 9, 1974.

[35] Schutz, B.F. "Networks of Gravitational Wave Detectors and Three Figures of Merit." *Classical and Quantum Gravity*, vol. 28, 125023, 2011.

[36] Anderson, W.G., et al. "A Powerful Tool for the Detection of Gravitational Waves." *Physical Review D*, vol. 63, 042003, 2001.

[37] Will, C.M. "The Confrontation between General Relativity and Experiment." *Living Reviews in Relativity*, vol. 17, 2014.

[38] Phinney, E.S. "A Practical Theorem on Gravitational Wave Backgrounds." *Astrophysical Journal Letters*, vol. 380, L17, 1991.

[39] Buonanno, A., and Damour, T. "Effective One-body Approach to General Relativistic Two-body Dynamics." *Physical Review D*, vol. 59, 084006, 1999.

[40] Misner, C.W., et al. "Gravitation." *W.H. Freeman*, 1973.

[41] Cutler, C., and Flanagan, E.E. "Gravitational Waves from Merging Compact Binaries: How Accurately Can One Extract the Binary's Parameters from the Inspiral Waveform?" *Physical Review D*, vol. 49, 2658, 1994.

[42] Finn, L.S. "Detection, Measurement, and Gravitational Radiation." *Physical Review D*, vol. 46, 5236, 1992.

[43] Allen, B. "Stochastic Gravitational-wave Backgrounds: A Search Strategy." *Physical Review D*, vol. 71, 062001, 2005.

[44] Abadie, J., et al. "Search for Gravitational Waves from Low Mass Compact Binary Coalescence in 186 Days of LIGO's Fifth Science Run." *Physical Review D*, vol. 80, 047101, 2009.

[45] Goodfellow, I., et al. "Deep Learning." *MIT Press*, 2016.

[46] LeCun, Y., Bengio, Y., and Hinton, G. "Deep Learning." *Nature*, vol. 521, pp. 436-444, 2015.

[47] Krizhevsky, A., et al. "ImageNet Classification with Deep Convolutional Neural Networks." *Communications of the ACM*, vol. 60, no. 6, pp. 84-90, 2017.

[48] Hochreiter, S., and Schmidhuber, J. "Long Short-Term Memory." *Neural Computation*, vol. 9, no. 8, pp. 1735-1780, 1997.

[49] Vaswani, A., et al. "Attention is All You Need." *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[50] He, K., et al. "Deep Residual Learning for Image Recognition." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770-778, 2016.

[51] Chollet, F. "Xception: Deep Learning with Depthwise Separable Convolutions." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1251-1258, 2017.

[52] Kingma, D.P., and Welling, M. "Auto-Encoding Variational Bayes." *arXiv preprint arXiv:1312.6114*, 2013.

[53] Radford, A., et al. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks." *arXiv preprint arXiv:1511.06434*, 2015.

[54] Silver, D., et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search." *Nature*, vol. 529, pp. 484-489, 2016.

[55] Sutskever, I., et al. "Sequence to Sequence Learning with Neural Networks." *Advances in Neural Information Processing Systems*, vol. 27, 2014.

[56] Devlin, J., et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, vol. 1, pp. 4171-4186, 2019.

[57] Dosovitskiy, A., et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale." *arXiv preprint arXiv:2010.11929*, 2020.

[58] Chen, T., et al. "A Simple Framework for Contrastive Learning of Visual Representations." *International Conference on Machine Learning*, pp. 1597-1607, 2020.

[59] Goodfellow, I., et al. "Generative Adversarial Nets." *Advances in Neural Information Processing Systems*, vol. 27, 2014.

[60] Bahdanau, D., et al. "Neural Machine Translation by Jointly Learning to Align and Translate." *International Conference on Learning Representations (ICLR)*, 2015.

[61] Ren, S., et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." *Advances in Neural Information Processing Systems*, vol. 28, 2015.

[62] Szegedy, C., et al. "Going Deeper with Convolutions." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1-9, 2015.

[63] Hinton, G.E., and Salakhutdinov, R.R. "Reducing the Dimensionality of Data with Neural Networks." *Science*, vol. 313, no. 5786, pp. 504-507, 2006.

[64] Simonyan, K., and Zisserman, A. "Very Deep Convolutional Networks for Large-Scale Image Recognition." *International Conference on Learning Representations (ICLR)*, 2015.

[65] Ioffe, S., and Szegedy, C. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." *International Conference on Machine Learning*, pp. 448-456, 2015.

[66] Krizhevsky, A., and Hinton, G.E. "Learning Multiple Layers of Features from Tiny Images." *Technical Report*, University of Toronto, 2009.

[67] Zeiler, M.D., and Fergus, R. "Visualizing and Understanding Convolutional Networks." *European Conference on Computer Vision*, pp. 818-833, 2014.

[68] Mikolov, T., et al. "Efficient Estimation of Word Representations in Vector Space." *arXiv preprint arXiv:1301.3781*, 2013.

[69] Schmidhuber, J. "Deep Learning in Neural Networks: An Overview." *Neural Networks*, vol. 61, pp. 85-117, 2015.

[70] Russakovsky, O., et al. "ImageNet Large Scale Visual Recognition Challenge." *International Journal of Computer Vision*, vol. 115, pp. 211-252, 2015.

[71] He, K., et al. "Mask R-CNN." *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 2980-2988, 2017.

[72] Zoph, B., et al. "Learning Transferable Architectures for Scalable Image Recognition." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 8697-8710, 2018.

[73] Brown, T.B., et al. "Language Models are Few-Shot Learners." *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[74] Girshick, R., et al. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 580-587, 2014.

[75] Tieleman, T., and Hinton, G. "Lecture 6.5 - RMSProp: Divide the Gradient by a Running Average of its Recent Magnitude." *Coursera: Neural Networks for Machine Learning*, 2012.

[76] Li, Y., et al. "ResNet in ResNet: Generalizing Residual Architectures with Non-Local Attention." *arXiv preprint arXiv:1912.09505*, 2019.

[77] Xu, B., et al. "Empirical Evaluation of Rectified Activations in Convolutional Network." *arXiv preprint arXiv:1505.00853*, 2015.

[78] Deng, J., et al. "ImageNet: A Large-Scale Hierarchical Image Database." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 248-255, 2009.

[79] Russell, S., and Norvig, P. "Artificial Intelligence: A Modern Approach." *Prentice Hall*, 4th edition, 2020.

[80] Goodfellow, I., et al. "Deep Learning." *MIT Press*, 2016.

[81] Silver, D., et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search." *Nature*, vol. 529, pp. 484-489, 2016.

[82] Hutter, F. "Automated Machine Learning: Methods, Systems, Challenges." *Springer*, 2019.

[83] McCarthy, J., et al. "A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence." *AI Magazine*, vol. 27, no. 4, pp. 12-14, 2006.

[84] Mnih, V., et al. "Human-Level Control through Deep Reinforcement Learning." *Nature*, vol. 518, pp. 529-533, 2015.

[85] LeCun, Y., et al. "A Theoretical Framework for Back-Propagation." *Proceedings of the 1988 Conference on Connectionist Models Summer School*, 1988.

[86] Bengio, Y., et al. "Learning Deep Architectures for AI." *Foundations and Trends in Machine Learning*, vol. 2, no. 1, pp. 1-127, 2009.

[87] Ng, A.Y., and Jordan, M.I. "On Discriminative vs. Generative Classifiers: A Comparison of Logistic Regression and Naive Bayes." *Advances in Neural Information Processing Systems*, vol. 14, 2002.

[88] Newell, A., and Simon, H.A. "GPS, a Program that Simulates Human Thought." *IEEE Transactions on Information Theory*, vol. 2, pp. 38-46, 1956.

[89] Turing, A.M. "Computing Machinery and Intelligence." *Mind*, vol. 59, no. 236, pp. 433-460, 1950.

A Appendix

A.1 GAN Generated Data Visualization

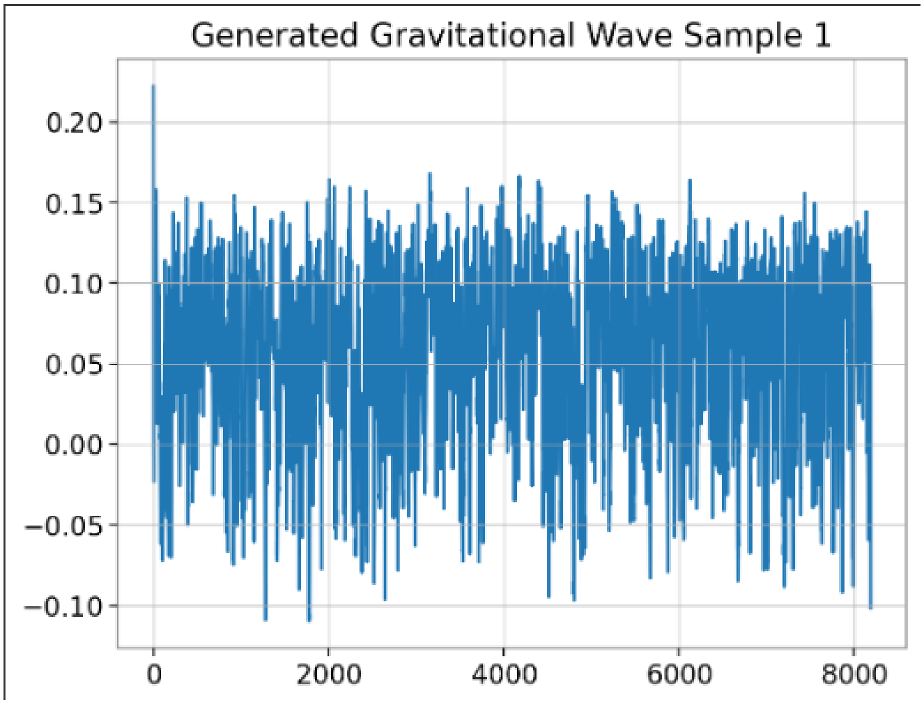

Figure 74: Example 1 of the GW segments generated.

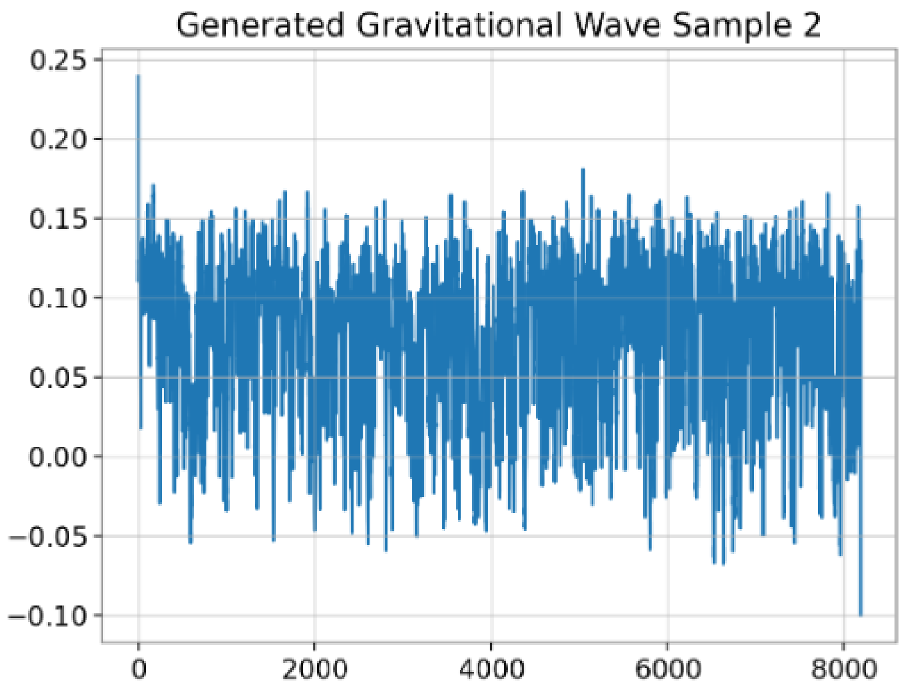

Figure 75: Example 2 of the GW segments generated.

A.2 WaveNet Generated Data Visualization

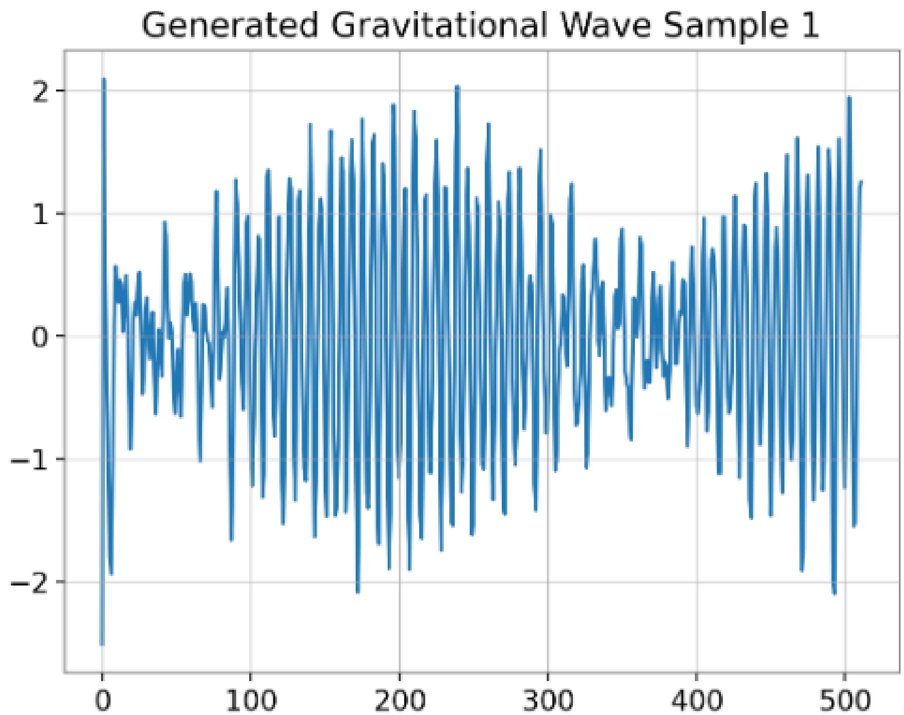

Figure 76: Example 1 of the GW segments generated.

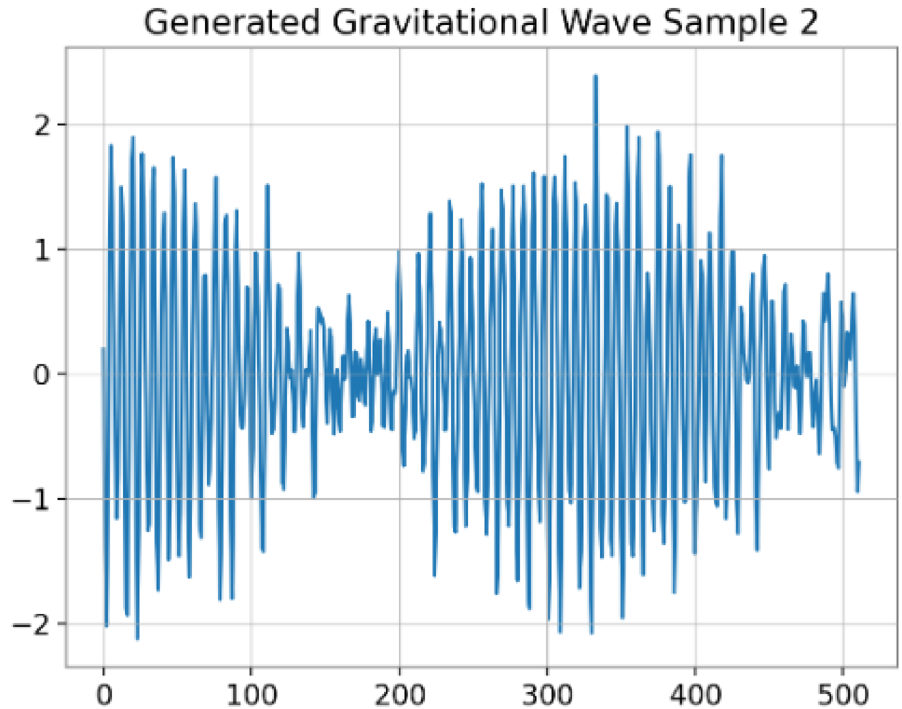

Figure 77: Example 2 of the GW segments generated.