Project Report

"Classification and Denoising of Gravitational Waves Using Deep Learning

Submitted In partial fulfillment of the requirements of the degree of

Bachelor of Technology (Electronics Engineering)

By

Mithil Joshi	181060032
Surbhi Singh	181061061
Tejas Kausal	181060036
Anagh Agrawal	181050007

Under the Guidance of

Prof. Dr. M. S. Panse

DEPARTMENT OF ELECTRICAL ENGINEERING
VEERMATA JIJABAI TECHNOLOGICAL INSTITUTE
MUMBAI-400019

(2018-2022)

DECLARATION OF THE STUDENT

We declare that this written submission represents our ideas in my own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources.

We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea / data / fact / source in our submission.

We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Signature of the students

Mithil Joshi	181060032
Surbhi Singh	181061061
Tejas Kausal	181060036
Anagh Agrawal	181050007

Date:

CERTIFICATE

The project report, "**Classification and Denoising of Gravitational Waves Using Deep Learning**" by **Mr. Mithil Joshi** (181060032), **Ms. Surbhi Singh** (181061061), **Mr. Tejas Kausal** (181060036) **and Mr. Anagh Agrawal** (181050007), is found to be satisfactory and is approved for partial fulfillment of the Degree of **B.Tech (Electronics)**.

Prof. Dr. M. S. Panse	**Prof. Dr. S. J. Bhosale**
(Project Guide)	**(Head of Electrical Engineering Dept)**

APPROVAL CERTIFICATE

The Project report, "**Classification and Denoising of Gravitational Waves Using Deep Learning**" by **Mr. Mithil Joshi** (181060032), **Ms. Surbhi Singh** (181061061), **Mr. Tejas Kausal** (181060036) **and Mr. Anagh Agrawal** (181050007), is found to be satisfactory and is approved for partial fulfillment of the Degree of **B.Tech (Electronics)**.

_____ _____

Prof. Dr. M. S. Panse **Examiner**

Project Guide

Department of Electrical Engineering

VJTI, Mumbai

Date:

Place:

ABSTRACT

Gravitational waves are perturbations on the flat background of space-time, analogous to the water waves on the otherwise flat ocean. These are caused by the most energetic processes in the Universe such as rotating neutron stars, colliding binaries, supernovae explosions and gravitational collapse in black holes. They are extremely weak so are very difficult to detect. G-Waves to carry information about astronomical phenomena through spacetime that can be detected on Earth. The Advanced Laser Interferometer Gravitational-Wave Observatory (Advanced LIGO), which has detectors in Livingston and Hanford, has directly observed these gravitational waves. The classical methods to identify the parameters which constitute the gravitational waves are time-consuming and computationally expensive. With the advent of Deep Learning and the development of dedicated processors, such computationally expensive problems have become efficient to solve. In this study, we create a convolutional neural network model which classifies the signal into noise and binary black hole merger signal. We take the gravitational waves measured by LIGO for validation of our model and further denoise the classified black hole merger signal using Denoising-Autoencoders.

Keywords: Gravitational waves, Convolutional Neural Network, Bi-Directional LSTMs, Denoising Autoencoders.

CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Background:

In the past, humans have observed the universe through different wavelengths of light but today, we use gravitational wave astronomy. Gravitational waves are the propagating perturbations in the otherwise tough, stiff fabric of space-time caused by energetic phenomena like rotating neutron stars, colliding binaries, supernovae explosions and gravitational collapse in black holes. Gravitational waves will allow us to look back into the history of the universe since the universe is transparent to the gravity moments long before light. Gravitational waves are carriers of motion information from the objects of the universe. Gravitational waves are not absorbed or reflected by matter. Today, with the United States' gravitational wave detector (LIGO) and its international partners, we are preparing to see the universe with a new set of eyes that do not depend on light.

Albert Einstein was the first to predict the existence of gravitational waves in his theory of general relativity. In his theory, he showed that a massive accelerating body will disrupt space-time and that would produce a wave in space-time that travels at the speed of light in all directions from the source.

The origin of gravitational waves are moving masses, but since the gravitational force is the weakest amongst the 4 natural forces, gravitational waves are exceedingly small. Waves of strength which can be detected on Earth will be produced by very massive systems undergoing large accelerations, like two orbiting black holes merging into one. Since systems like these are light-years away, the detection is of minute effects generated from most energetic astrophysical systems. By the time it reaches Earth, it loses its energy. It becomes the size of the width of atoms. That's why it's very difficult to detect it.

Scientists came in contact with the wave on 14th September 2015 at LIGO. The wave detected was caused by the collision of two black holes. This has given scientists the opportunity to explore and see the universe in a whole new way. These waves contain the information of its origin and tell us more about gravity itself. If the instruments are sensitive enough, it can catch

the waves that were produced at the time of the Big Bang and we will get the chance to know more about the creation of the universe.

Earlier scientists were using EM radiation (visible light, X-ray, radio waves, etc.) to study the universe. Many more methods were also used for it but would give a similar result. The Gravitational-wave is different from the EM wave. Gravitational waves interact weakly with matter, they travel through the universe unimpeded and give us clear information about its origin.

Laser Interferometer Gravitational-Wave Observatory (LIGO) is an 'L'-shaped instrument that measures the gravitational wave. It consists of lasers and mirrors. When the gravitational wave passes through the instrument, the position of the mirrors changes. This change in position is caused by the strain applied by the wave. It becomes a challenge to detect the gravitational wave due to various noises emerging from the instrument.

1.2 Motivation:

Gravitational wave astronomy will help explore some of the great questions in physics: How do black holes form? Is General Relativity the correct description of gravity? How does matter act under the extremes of temperature and pressure in neutron stars and supernovae?

However the traditional methods for detection of Gravitational waves such as Matched Filtering which is the most sensitive algorithm used by LIGO targets a 3D subspace of the 8D parameter space. Hence we needed a new paradigm to overcome the limitations and computational challenges of existing algorithms. An ideal candidate is the rapidly advancing field of Deep Learning, that can learn directly from raw data, without any manual feature engineering, in combination with optimization techniques based on back-propagation and gradient descent. Deep learning, especially with the aid of GPU computing, has recently achieved immense success in both commercial applications and artificial intelligence (AI) research.

The discovery and efficient detection of gravitational waves helps out immensely with the theory of Quantum Mechanics. We can develop a theory of quantum gravity, and once we have that, we can finally unify General Relativity and Quantum Mechanics into a Grand Unified Theory. And once we have this theory developed, we can harness the very processes the universe uses to function in order to create quantum Internet, efficient space travel, cryogenics, etc.

1.3 Objectives:

Our research work is based on Deep Learning with applications in Gravitational wave astronomy. Deep learning refers to Machine Learning algorithms based on neural networks having several hidden layers making it deep.

The idea of the project is creation of a Deep Learning model for the classification of Gravitational waves which are detected in real time by the Gravitational waves Detectors (eg. LOGO, VIRGO, etc.) based on its source i.e. into pure noise and Binary Black Hole Merger waves. The Noise signals can then be discarded and the BBH merger waves are further Denoised using Denoising Autoencoders to extract pure Binary Black Hole Merger signal.

CHAPTER 2: LITERATURE REVIEW

Paper 1: N. Lopac, F. Hržić, I. P. Vuksanović and J. Lerga, "Detection of Non-Stationary GW Signals in High Noise From Cohen's Class of Time–Frequency Representations Using Deep Learning," in *IEEE Access*, vol. 10, pp. 2408-2428, 2022, doi: 10.1109/ACCESS.2021.3139850.

Analysis of non-stationary signals in a noisy environment is a challenging research topic in many fields, often requiring simultaneous signal decomposition in the time and frequency domain. This paper proposes a method for the classification of noisy non-stationary time-series signals based on Cohen's class of their time-frequency representations (TFRs) and deep learning algorithms. We demonstrated the proposed approach on the example of detecting gravitational-wave (GW) signals in intensive real-life, non-stationary, non-white, and non-Gaussian noise. For this purpose, we prepared a dataset based on the actual data from the Laser Interferometer Gravitational-Wave Observatory (LIGO) detector and the synthetic GW signals obtained by realistic simulations. Next, 12 different TFRs from Cohen's class were calculated from the original noisy time-series data and used to train three state-of-the-art convolutional neural network (CNN) architectures: ResNet-101, Xception, and EfficientNet. The obtained classification results are compared to those achieved by the base model trained on the original time series. Analysis of the results suggests that the proposed approach combining deep CNN architectures with Cohen's class TFRs yields high values of performance metrics and significantly improves the classification performance compared to the base model. The TFR-CNN models achieve the values of the classification accuracy of up to 97.10%, the area under the receiver operating characteristic curve (ROC AUC) of up to 0.9885, the recall of up to 95.87%, the precision of up to 99.51%, the F1 score of up to 97.03%, and the area under the precision-recall curve (PR AUC) of up to 0.9920. This classification performance is obtained on the dataset in which the signal-to-noise ratio (SNR) values of the raw, noisy time-series signals range from −123.46 to −2.27 dB. Therefore, this study suggests that using alternative TFRs of Cohen's class can improve the deep learning-based detection of non-stationary GW signals in an intensive noise environment. Moreover, the proposed approach can also be a viable solution for deep learning-based analysis of numerous other noisy non-stationary signals in different practical

applications.

Paper 2: H. Shen, D. George, E. A. Huerta and Z. Zhao, "Denoising Gravitational Waves with Enhanced Deep Recurrent Denoising Auto-encoders," ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2019, pp. 3237-3241, doi: 10.1109/ICASSP.2019.8683061.

Denoising of time domain data is a crucial task for many applications such as communication, translation, virtual assistants etc. For this task, a combination of a recurrent neural net (RNNs) with a Denoising Auto-Encoder (DAEs) has shown promising results. However, this combined model is challenged when operating with low signal-to-noise ratio (SNR) data embedded in non-Gaussian and non-stationary noise. To address this issue, we design a novel model, referred to as `Enhanced Deep Recurrent Denoising Auto-Encoder' (EDR-DAE), that incorporates a signal amplifier layer, and applies curriculum learning by first denoising high SNR signals, before gradually decreasing the SNR until the signals become noise dominated. We showcase the performance of EDR-DAE using time-series data that describes gravitational waves embedded in very noisy backgrounds. In addition, we show that EDRDAE can accurately denoise signals whose topology is significantly more complex than those used for training, demonstrating that our model generalizes to new classes of gravitational waves that are beyond the scope of established denoising algorithms.

Paper 3: S. Fan, Y. Wang, Y. Luo, A. Schmitt and S. Yu, “Improving Gravitational Wave Detection with 2D Convolutional Neural Networks,” 2020 25th International Conference on Pattern Recognition (ICPR), 2021, pp. 7103-7110, doi: 10.1109/ICPR48806.2021.9412180

The direct observation to quickly detect faint GW signals from a large number of time series with background noise under unknown probability distributions signals remains a challenge. Traditional methods such as matched-filtering have high computational complexity. To avoid these weaknesses, one-dimensional (1D) Convolutional Neural Networks (CNNs) are introduced

to achieve fast online detection in milliseconds In this work, the input data is pre-processed to form a 2D spectrum by Short-time Fourier transform (STFT), where frequency features are extracted without learning. Then, carrying out two 1D convolutions across time and frequency axes respectively, and concatenating the time-amplitude and frequency-amplitude feature maps with equal proportion subsequently, the frequency and time features are treated equally as the input of our following two-dimensional CNNs. The simulation of our above ideas works on a generated data set with uniformly varying SNR (2-17), which combines the GW signal generated by PYCBC and the background noise sampled directly from LIGO.

Paper 4: F. U. M. Ullah, A. Ullah, I. U. Haq, S. Rho and S. W. Baik, "Short-Term Prediction of Residential Power Energy Consumption via CNN and Multi-Layer Bi-Directional LSTM Networks," in IEEE Access, vol. 8, pp. 123369-123380, 2020, doi: 10.1109/ACCESS.2019.2963045.

Excessive Power Consumption (PC) and demand for power is increasing on a daily basis, due to advancements in technology, the rise in electricity-dependent machinery, and the growth of the human population. We present an intelligent hybrid technique that combines a Convolutional Neural Network (CNN) with a Multi-layer Bi-directional Long-short Term Memory (M-BDSM) method using three steps. When applied to short-term power ECP, this approach helps to provide efficient power management i.e. it can assist the supplier to produce the optimum amount of power. The first step in our proposed method integrates the pre-processing and data organization mechanisms to refine the data and remove abnormalities. The second step employs a deep learning network, where the sequence of refined data is fed into the CNN via the M-BD LSTM network to learn the sequence pattern effectively. The third step generates the ECP/PC by comparing actual and predicted data series and evaluates the prediction using error metrics. The proposed method achieves better prediction results than existing techniques, thus demonstrating its effectiveness. Furthermore, it achieved the smallest value of the Mean Square Error (MSE) and Root Mean Square Error (RMSE) for individual household dataset using 10-fold Cross Validation (CV) and a hold-out (CV) method.

Paper 5: H. Chiang, Y. Hsieh, S. Fu, K. Hung, Y. Tsao and S. Chien, "Noise Reduction in ECG Signals Using Fully Convolutional Denoising Autoencoders," in IEEE Access, vol. 7, pp. 60806-60813, 2019, doi: 10.1109/ACCESS.2019.2912036.

In real-world scenarios, ECG signals are prone to be contaminated with various noises, which may lead to wrong interpretation. Therefore, significant attention has been paid on denoising of ECG for accurate diagnosis and analysis. A denoising autoencoder (DAE) can be applied to reconstruct clean data from its noisy version. In this paper, a DAE using a fully convolutional network (FCN) is proposed for ECG signal denoising. Meanwhile, the proposed FCN-based DAE can perform compression with regard to the DAE architecture. The denoising performance is evaluated using root mean square error (RMSE), percentage root mean square difference (PRD), and improvement in signal-to-noise ratio (SNRimp). The results of the experiments conducted on noisy ECG signals of different levels of input SNR show that the FCN acquires better performance as compared to deep fully connected neural network based and convolutional neural network based denoising models.

Paper 6: S. Kiranyaz, T. Ince and M. Gabbouj, "Real-Time Patient-Specific ECG Classification by 1-D Convolutional Neural Networks," in IEEE Transactions on Biomedical Engineering, vol. 63, no. 3, pp. 664-675, March 2016, doi: 10.1109/TBME.2015.2468589.

This paper presents a fast and accurate patient-specific electrocardiogram (ECG) classification system. Methods: An adaptive implementation of 1D Convolutional Neural Networks (CNNs) is inherently used to fuse the two major blocks of the ECG classification into a single learning body: feature extraction and classification. Therefore, feature extraction ability can further improve the classification performance. Since this also negates the necessity to extract hand-crafted manual features, once a dedicated CNN is trained for a particular patient, it can solely be used to classify possibly long ECG data stream in a fast and accurate manner or alternatively, such a solution can conveniently be used for real-time ECG monitoring and early alert system on a light-weight wearable device. This paper was used to understand feature

extraction and classification performed by 1D CNN for the purpose of our application.

Paper 7: G. Baltus, J. -R. Cudell, J. Janquart, M. Lopez, S. Caudill and A. Reza, "Detecting the early inspiral of a gravitational-wave signal with convolutional neural networks," 2021 International Conference on Content-Based Multimedia Indexing (CBMI), 2021, pp. 1-6, doi: 10.1109/CBMI50038.2021.9461919.

This paper introduces a novel methodology for the operation of an early alert system for gravitational waves. It is based on short convolutional neural networks. We focus on compact binary coalescences, for light, intermediate and heavy binary-neutron star systems. The signals are 1-dimensional time series − the whitened time-strain − injected in Gaussian noise built from the power-spectral density of the LIGO detectors at design sensitivity. It builds short 1-dimensional convolutional neural networks to detect these types of events by training them on part of the early inspiral.

Paper 8: A. Majumdar, "Blind Denoising Autoencoder," in IEEE Transactions on Neural Networks and Learning Systems, vol. 30, no. 1, pp. 312-317, Jan. 2019, doi: 10.1109/TNNLS.2018.2838679.

The term “blind denoising” refers to the fact that the basis used for denoising is learned from the noisy sample itself during denoising. Dictionary learning- and transform learning-based formulations for blind denoising are well known. But there has been no autoencoder-based solution for the said blind denoising approach. So far, autoencoder-based denoising formulations have learned the model on a separate training data and have used the learned model to denoise test samples. Such a methodology fails when the test image (to denoise) is not of the same kind as the models learned with. This will be the first work, where we learn the autoencoder from the noisy sample while denoising. Experimental results show that our proposed method performs better than dictionary learning (K-singular value decomposition), transform learning, sparse stacked denoising autoencoder, and the gold standard BM3D algorithm.

CHAPTER 3: GRAVITATIONAL WAVES

3.1 Gravitational Waves:

Ripples in space-time caused by the motion of objects in the universe. The strongest gravitational waves are produced by cataclysmic events such as colliding black holes, supernovae (massive stars exploding at the end of their lifetimes), and colliding neutron stars. Other waves are predicted to be caused by the rotation of neutron stars that are not perfect spheres, and possibly even the remnants of gravitational radiation created by the Big Bang.

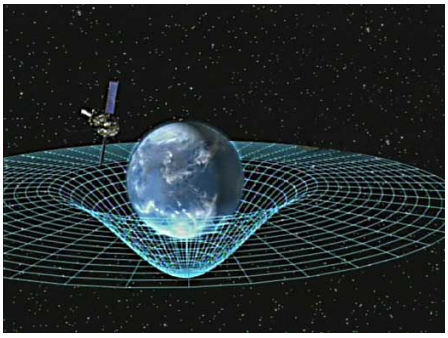


Figure 3.1 Effect of Mass on Space Time Curvature

3.2 Origin of Gravitational Waves:

Gravitational waves are produced by masses moving through space-time in a special way. The simplest system that produces gravitational waves is two masses orbiting their common center of mass. One of the most common such systems is a binary star system – two stars orbiting each other's common center of mass. You can have any combination of a normal star, giant star, white dwarf, neutron star, or black hole. Another place you might find large masses orbiting each other is the center of a galaxy - if two galaxies merged, their central supermassive black holes would orbit for a long time before they also merged.

As the black holes, stars, or galaxies orbit each other, they send out waves of "gravitational radiation" that travel at the speed of light. The waves that reach us are extremely weak. Despite being weak, the waves can travel unobstructed within the 'fabric' of space-time, providing us with information that light cannot.

3.3 Detection of Gravitational Waves:

The detection of gravitational waves requires measurements that detect changes in distance less than the size of an atomic nucleus. Scientists use interferometry, which consists of two parts: test masses separated by a distance and lasers to measure that distance. Test masses are set at a large distance from each other – the large distance helps make any change in their distance be large enough to measure. The masses are then shielded from all disturbances except gravity, which we cannot shield against. Then lasers make continuous measurements of the distance between each of the test masses. The masses are free to move in response to gravity so that when a gravitational wave passes, space-time stretches and the time it takes light to travel between the masses changes.

Gravitational waves were first detected by the ground-based LIGO detectors in 2015 using ground-based facilities in Washington and Louisiana. The Virgo detector in Italy spied its first gravitational wave in 2017. A space-based observatory called LISA is scheduled to launch in the early 2030s as part of the European Space Agency's Cosmic Visions Program. The source on which we focus in this project is Compact Binary Coalescence (CBC).

3.4 Compact Binary Coalescence Gravitational Waves:

So far, all of the objects LIGO has detected fall into this category. Compact binary inspiral gravitational waves are produced by orbiting pairs of massive and dense ("compact") objects like white dwarf stars, black holes, and neutron stars. There are three subclasses of "compact binary" systems in this category of gravitational-wave generators:

- Binary Neutron Star (BNS)
- Binary Black Hole (BBH)
- Neutron Star-Black Hole Binary (NSBH)

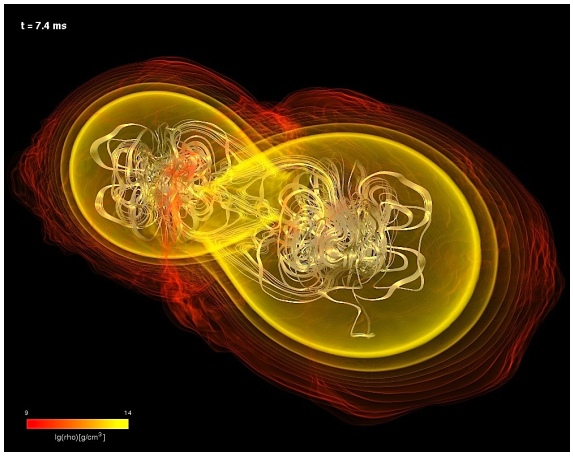


Figure 3.2 Merging of Two CBC Masses

Each binary pair creates a unique pattern of gravitational waves, but the mechanism of wave-generation is the same across all three called Inspiral.

Inspiral occurs over millions of years as pairs of dense compact objects revolve around each other. As they orbit, they emit gravitational waves that carry away some of the system's orbital energy. As a result, over eons, the objects orbit closer and closer together. Unfortunately, moving closer causes them to orbit each other faster, which causes them to emit stronger gravitational waves, which causes them to lose more orbital energy.

3.5 LIGO:

LIGO stands for "Laser Interferometer Gravitational-wave Observatory". It is the world's largest gravitational wave observatory and a marvel of precision engineering. Comprising two enormous laser interferometers located 3000 kilometers apart, LIGO exploits the physical properties of light and of space itself to detect and understand the origins of gravitational waves (GW).

Though its mission is to detect gravitational waves from some of the most violent and energetic processes in the Universe, the data LIGO collects may have far-reaching effects on many areas of physics including gravitation, relativity, astrophysics, cosmology, particle physics, and nuclear physics.

3.6 Interferometer:

Interferometers work by merging two or more sources of light to create an interference pattern, which can be measured and analyzed. They are often used to make very small measurements that are not achievable any other way. This is why they are so powerful for detecting gravitational waves--LIGO's interferometers are designed to measure a distance 1/10,000th the width of a proton!

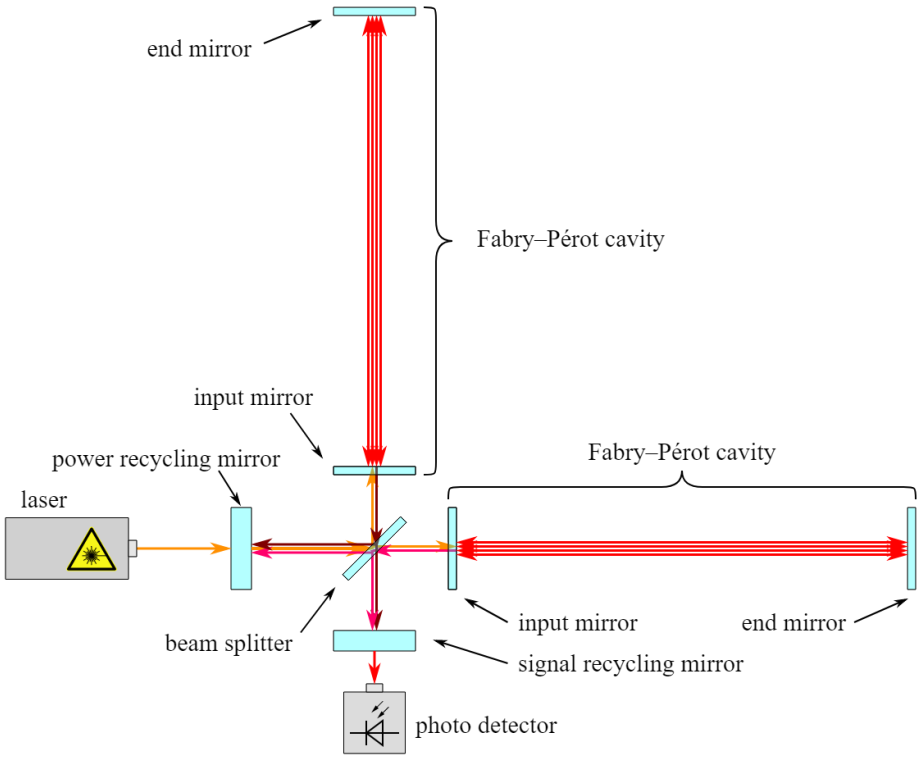


Figure 3.3 Interferometer

3.7 Sources Of CBC focussed in this project:

1. **Black hole binaries :** Black hole binaries emit gravitational waves during their in-spiral, merger, and ring-down phases. The first direct detection of gravitational waves, GW150914, came from the merger of two black holes.

2. **Spinning neutron stars :** A spinning neutron star will generally emit no gravitational radiation because neutron stars are highly dense objects with a strong gravitational field that keeps them almost perfectly spherical. In some cases, however, there might be slight deformities on the surface called "mountains", which are bumps extending no more than 10 centimeters (4 inches) above the surface, that make the spinning spherically asymmetric. This gives the star a quadrupole moment that changes with time, and it will emit gravitational waves until the deformities are smoothed out.

3.8 Properties and Behavior of Gravitational Waves:

1. **Energy, momentum, and angular momentum** : Gravitational waves are able to carry energy, momentum, and angular momentum and by doing so they carry those away from the source.. Thus, for example, a binary system loses angular momentum as the two orbiting objects spiral towards each other—the angular momentum is radiated away by gravitational waves.

2. **Significance for study of the early universe :** Due to the weakness of the coupling of gravity to matter, gravitational waves experience very little absorption or scattering, even as they travel over astronomical distances. In particular, gravitational waves are expected to be unaffected by the opacity of the very early universe. In these early phases, space had not yet become "transparent", so observations based upon light, radio waves, and other electromagnetic radiation that far back into time are limited or unavailable. Therefore, gravitational waves are expected in principle to have the potential to provide a wealth of observational data about the very early universe.
3. **Determining direction of travel :** The difficulty in directly detecting gravitational waves means it is also difficult for a single detector to identify by itself the direction of a source. Therefore, multiple detectors are used, both to distinguish signals from other "noise" by confirming the signal is not of earthly origin, and also to determine direction by means of triangulation.

CHAPTER 4: DEEP LEARNING

In the past few years, artificial intelligence (AI) has been a subject of intense media hype. Machine learning, deep learning, and AI come up in countless articles, often outside of technology-minded publications. We're promised a future of intelligent chatbots, self-driving cars, and virtual assistants—a future sometimes painted in a grim light and other times as utopian, where human jobs will be scarce and most economic activity will be handled by robots or AI agents.

4.1 Artificial Intelligence

A concise definition of the field would be as follows: the effort to automate intellectual tasks normally performed by humans. As such, AI is a general field that encompasses machine learning and deep learning.

4.2 Deep Learning

Deep learning is a specific subfield of machine learning: a new take on learning representations from data that puts an emphasis on learning successive layers of increasingly meaningful representations. In deep learning, these layered representations are (almost always) learned via models called neural networks, structured in literal layers stacked on top of each other. Deep learning is a mathematical framework for learning representations from data.

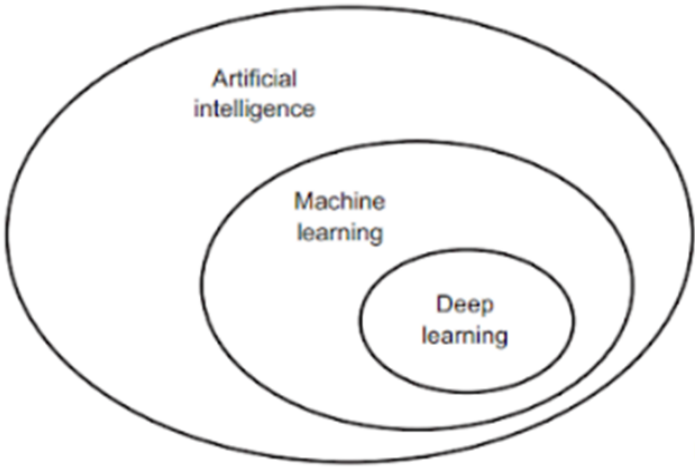


Figure 4.1 Subfields of AI

4.3 Convolutional Neural Network

A Convolutional Neural Network, specifically, is a deep feed-forward network that extends a "classic" artificial neural network by adding more layers (deep learning), including the introduction of convolutional blocks. The term "convolutional" comes from the implementation of convolution blocks in the network. The typical structure of a CNN is composed of three layers: convolutional, pooling, and fully connected layers. These layers make use of several components/techniques, such as convolutions, activation functions, pooling, dropout, batch normalization, fully connected blocks, among others, that can be combined in multiple ways.

The input data can have a 1D, 2D, or 3D format. We have used 1-D data. There are several advantages in using 1D CNNs, such as good performance on a limited amount of data, low computational complexity, faster training process, and a good ability to extract relevant features in sequence data and time-series. The convolutional layers are responsible for processing the feature extraction tasks. It processes the extraction by applying convolution operations to the input, and the convolution result is fed to the next layer's input. The convolutions operations are defined by several filters, a kernel size, a padding, and a stride, which finally generates a feature map after applying an activation function, such as ReLU or Tanh function.

Components –

- Kernel: In each of the convolution layers, the input is run/slid by a kernel. A kernel consists of mathematical operations that produce a matrix dot product called an activation map.
- Padding: Padding is a technique that avoids the loss of information on the input data borders, caused by the kernel operations, and consists of adding zeros around the input margins.
- Activation Function: The activation functions are responsible for applying nonlinearity to the models, through the application of the operations on activation maps, making it possible to incorporate complex input relations. In our code we have used ReLU and Softmax.

- Pooling: Regarding the pooling operations, the goal is always to reduce the spatial size of the convolved features and avoid overfitting.
- Dropout: Dropout is a technique used for avoiding overfitting, which can be applied to fully connected or convolutional layers.

We will be using 1D convolutional Neural Networks for our application of classifying the gravitational wave data. The classifier will be using 1D CNN. Our input is a time series data where the kernel will move in one dimension that's why 1D.

In CNN, we slide the kernel over the input. The elements of the kernel get multiplied by the corresponding elements of the time series that they cover at a given point. Then the results of the multiplication are added together and a nonlinear activation function is applied to the value. The resulting value becomes an element of a new "filtered" univariate time series, and then the kernel moves forward along the time series to produce the next value. The number of new "filtered" time series is the same as the number of convolution kernels. Depending on the length of the kernel, different aspects, properties, "features" of the initial time series get captured in each of the new filtered series.

The next step is to apply global max-pooling to each of the filtered time series vectors: the largest value is taken from each vector. A new vector is formed from these values, and this vector of maxima is the final feature vector that can be used as an input to another convolution layer or to a regular fully connected layer. The feature vector captures patterns from the input. In deep networks, the first few feature vectors capture finer patterns from the signal and the next feature vector generalizes over the pattern of the signal.

Our model consists of a classifier which classifies the given input as signal or noise. The noise will be discarded whereas the signal will be used for denoising.

4.4 Long Short Term Memory (LSTM)

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task. Unfortunately,as the gap between the relevant information and the point where it is needed grows , RNNs become unable to learn to connect the information. In theory, RNNs are capable of handling such "long-term dependencies." but in practice, RNNs don't seem to be able to learn them.

Long Short Term Memory networks are a special kind of RNN, capable of learning long-term

dependencies. LSTMs , like RNNs, have a chain-like structure, of repeating modules each having four neural network layer

The LSTM has the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through.

There are three types of gates:

- Forget gate – controls how much information the memory cell will receive from the memory cell from the previous step
- Update (input) gate – decides whether the memory cell will be updated. Also, it controls how much information the current memory cell will receive from a potentially new memory cell.
- Output gate – controls the value of the next hidden state

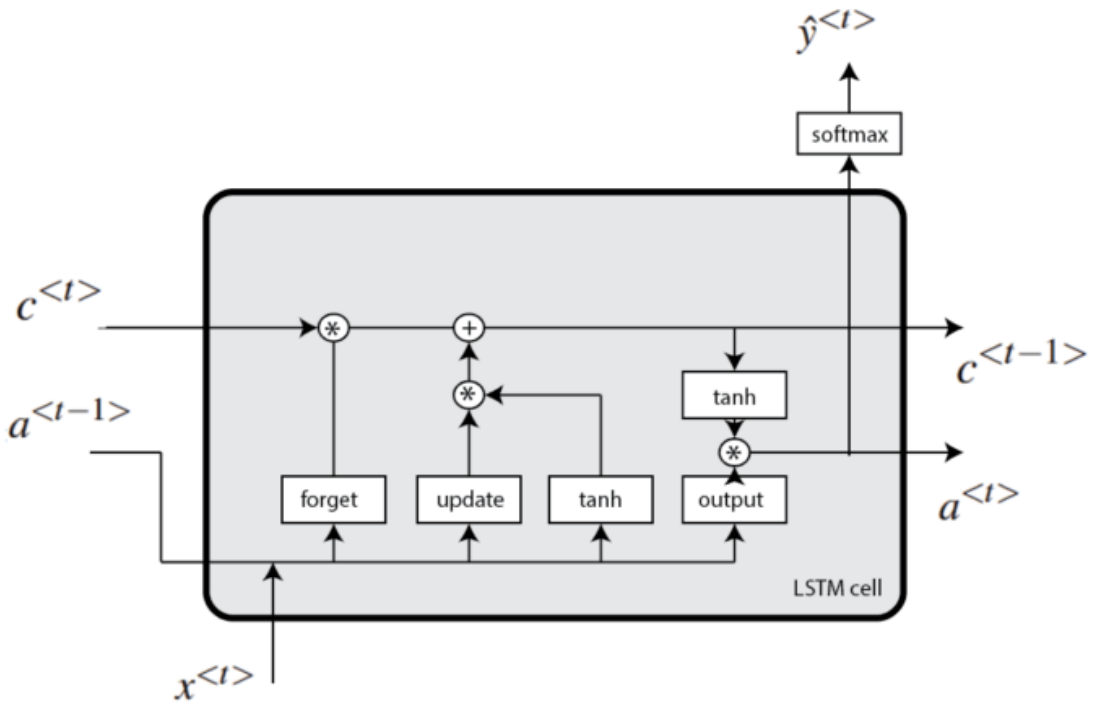


Figure 4.2 LSTM Architecture

Bidirectional LSTM (BiLSTM) is a variant of standard LSTM where the input flows in both directions, and it's capable of utilizing information from both sides. BiLSTM adds one more LSTM layer, which reverses the direction of information flow. It means that the input sequence flows backward in the additional LSTM layer. Then the outputs from both LSTM layers are combined in several ways, such as average, sum, multiplication, or concatenation.

This type of architecture has many advantages in real-world problems since every component of an input sequence has information from both the past and present. For this reason, BiLSTM can produce a more meaningful output, combining LSTM layers from both directions.

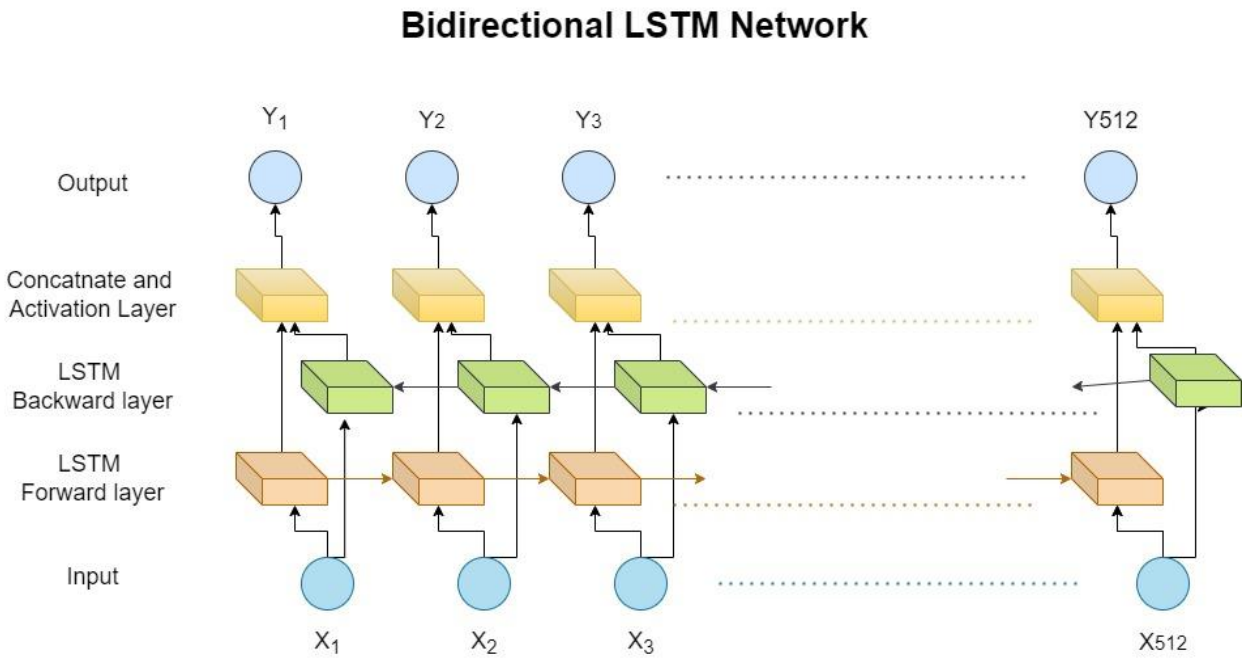


Figure 4.3 Bidirectional LSTM Network

4.5 Curriculum Learning

Curriculum learning (CL) is a training strategy that trains a machine learning model from easier data to harder data, which imitates the meaningful learning order in human curricula. As an easy-to-use plug-in, the CL strategy has demonstrated its power in improving the generalization capacity and convergence rate of various models in a wide range of scenarios such as computer vision and natural language processing etc.

Convex learning is invariant to the order of sample presentation, but both human learning and ConvNets are not: the order in which things are learned is significant. Usually, in any formal academic setting, the information to be taught to students is carefully structured in a curriculum, so that "easier" concepts are introduced first, and further knowledge is systematically acquired by mastering concepts with increasing difficulty. The idea of training neural networks with a curriculum can be traced back to Elman [32]. The experiment involved learning simple grammar with a recurrent network. Elman noted that networks who start "small" (with limited working memory) succeed in the task better than "adultlike" networks who are exposed to the full grammar at their maximum learning capacity.

CHAPTER 5: SYSTEM DEVELOPMENT

5.1 Schematic Diagram:

5.1.1 Final Project Schematic:

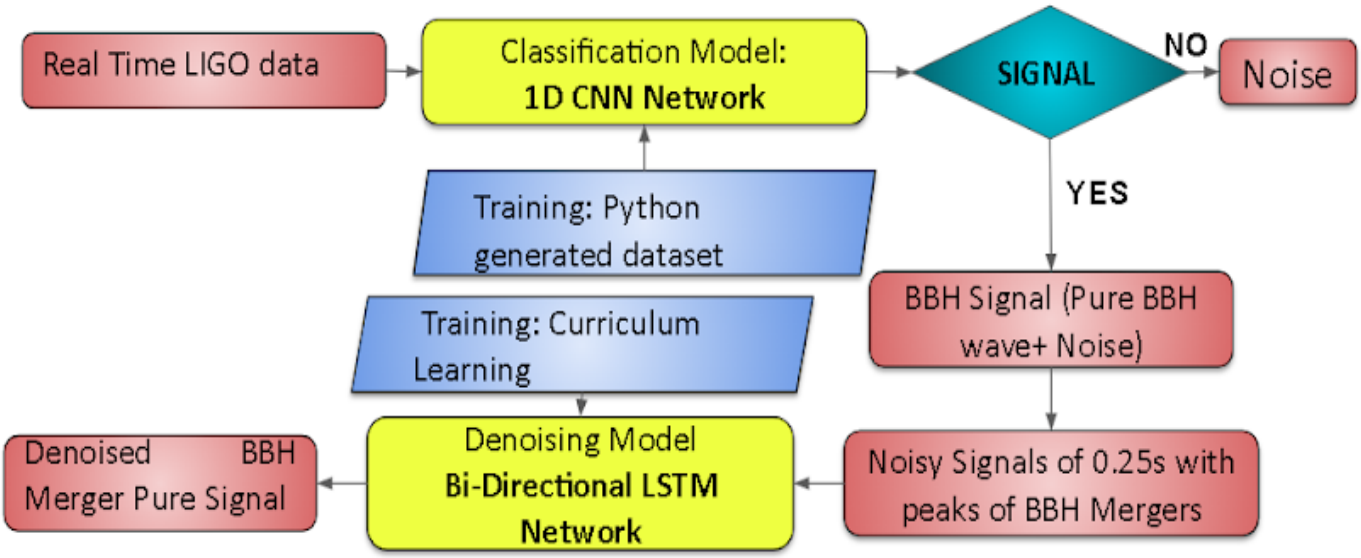


Figure 5.1 Final Project Schematic

Description:

1. We first went through numerous papers for our project
2. Out of all the 3 Gravitational wave detectors, the project focuses on LIGO detector signals.
3. The data for training, testing and validation was generated using the python package PyCBC.
4. The real time signal extracted from the LIGO detector, is inputted into the Trained Classification Model.
5. The classification model architecture is developed using 1D Convolutional Neural Network Layers.
6. The classification model categorizes the signal as either Pure Noise or Binary Black Hole (BBH) Merger Wave. The noise signal is discarded.
7. To extract the pure signal from the Noisy BBH merger wave, it has to be processed through the Denoising Model.

8. The denoising model architecture is developed using Bi-Directional LSTM layers. The training of the architecture is done using simulated waves and curriculum learning using SNR of pure BBH signal embedded in noise as the parameter.
9. The signal is sliced to 0.25 around the peak of Mergers.
10. The signal is sent through the Denoised Model, which outputs pure Binary Black Hole Merger waves and eliminates the noise.
11. Thus the network classifies and denoises the data, without significant loss of original data and protects the valuable information which can be extracted from the signal using further processing.

5.1.2 Signal Classification Schematic:

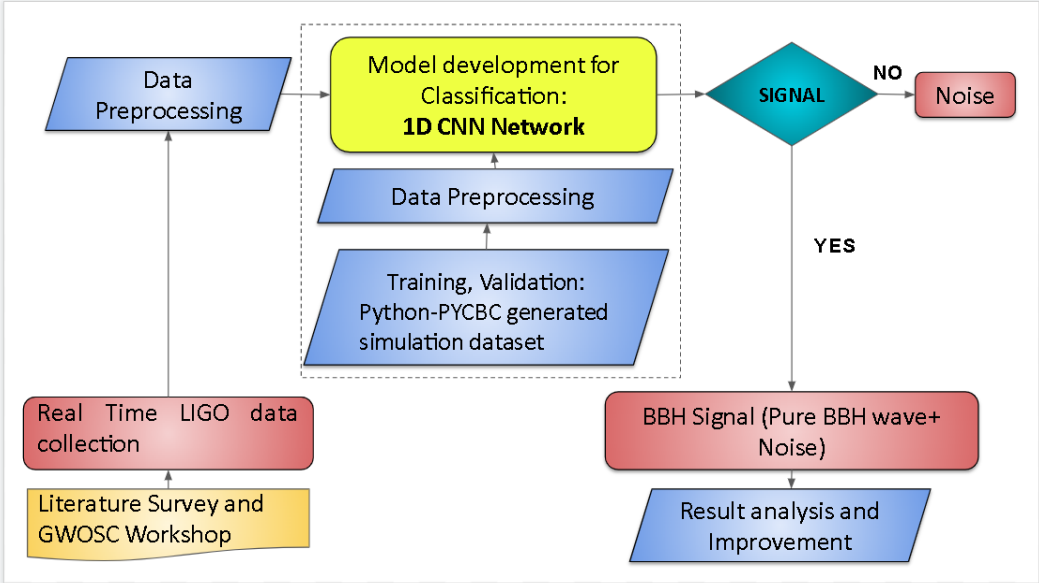


Figure 5.2 Signal Classification Schematic

Description:

1. There are three gravitational detectors, out of which we will be working on the signals detected by the LIGO detector.

2. Since there were less number of real time detections, hence we were short of real time data to train our neural network. Hence we decided to go for simulated waves as our training data.
3. The software which we will be using to generate our data is PYCBC, which is mostly used worldwide to simulate gravitational waves.
4. The sample in the data can be either pure gaussian Noise or Pure Signal+Noise (Signal superimposed on Noise).
5. Since our data is 1-dimensional hence we will be developing a 1D CNN to classify the data as black hole generated waves or simple noise.
6. Noise waves will be discarded, while Binary Black Hole(BBH) waves can be sent for further processing.

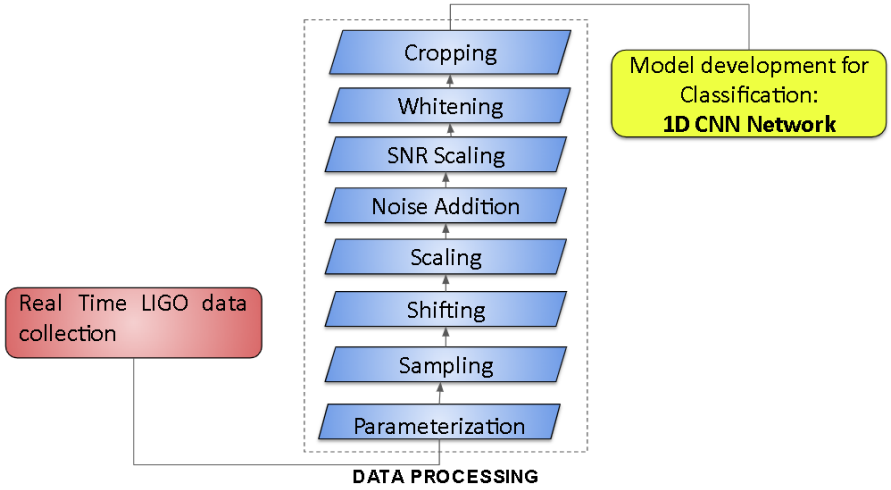


Figure 5.3 Data Preprocessing Pipeline

Description:

1. We have shown the internal workings of the "Data Preprocessing" block.
2. The preprocessing steps involve:
    a. **Parameterization:** 8 parameters were selected for characterization of Binary Black Hole Merger waves. These parameters were - approximant, mass_1, mass_2, spin_z_axis_1, spin_z_zxis_2, sampling_time, lower_frequency and distance between merger & detector.
    b. **Sampling:** the sampling frequency was chosen to be 2048 Hz and the

signal length was 8s, hence total timesteps were 16384.

c. **Shifting:** The pure signal was shifted such that the merger (signified by the highest frequency) lies between 6 to 8s.

d. **Scaling:** The distance between the merger & detector was kept between 2500m & 3000m so that the order of the signal was 10^{-22}.

e. **Noise addition:** The White Additive Gaussian noise of order 10^{-22} was added to the signal.

f. **SNR Scaling:** The signal-to-noise ratio was scaled between 2 and 17 for the signals using the process called matched filtering.

g. **Whitening:** We applied whitening to noisy signals to flatten the Power spectrum which pushes its correlation closer to white. The whitened signal is easier to decode using ML detection and estimation formulations.

h. **Cropping:** The final operation is cropping the signal such that it comes down to 8s and 16384 samples.

3. This preprocessed data is then used as the input to the deep learning model.

5.1.3 Signal Denoising Schematic:

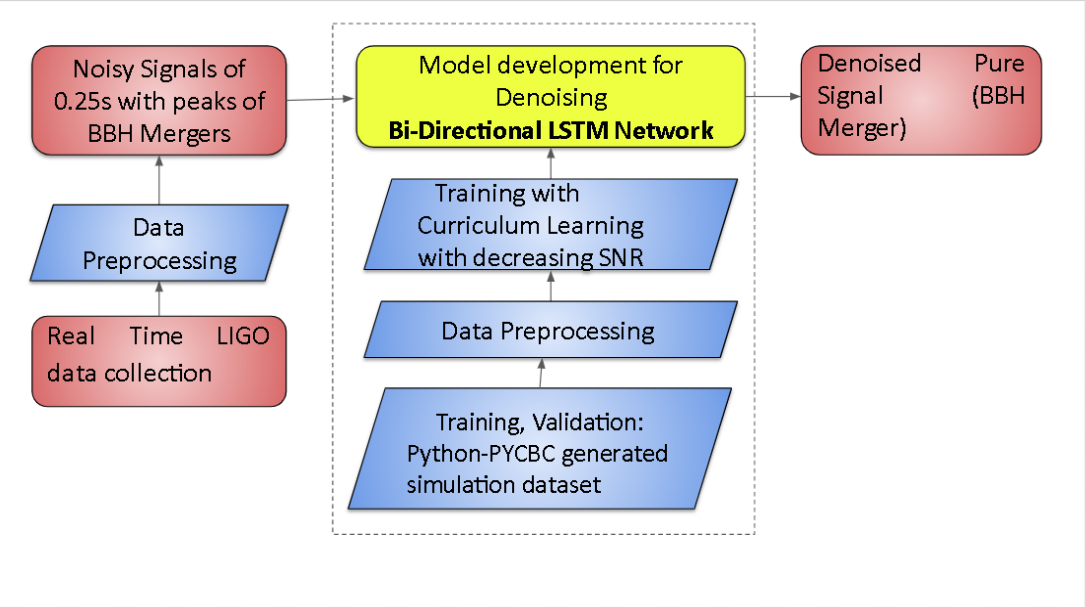


Figure 5.4 Signal Denoising Schematic

Description:

1. We collected the real-time gravitational waves from the LIGO detector. During the detection process, the pure signal gets embedded in noise, from which we want to extract the pure signal.
2. We applied the same data preprocessing algorithms on the detected signals and cropped their lengths to 0.25s i.e. 512 samples at the sampling rate of 2048 Hz.
3. For training the LSTM Denoising model, we simulated the training set using the PyCBC package in python and applied the same preprocessing operations.
4. We trained the network using the concept of Curriculum Learning wherein the data is inputted such that after every 10 epochs the SNR decreases by 4 units.
5. Then the model was validated and tested on a dataset simulated by the PyCBC package.
6. The model was tested for real-time signals and the output was Denoised Pure Signal (BBH merger).

5.2 Collection Of Dataset

There are numerous sources of Gravitational waves, but this project focuses on a particular source which are the Binary Black Hole Wave Mergers, which come under the technical categorization of "Colliding Binary Coalescence (CBC)." Hence there are two classes of waves that are simulated i.e. Binary Black Hole Coalescence + Noise and Pure Noise signals.

5.2.1 Classification:

Input data specifications: -

- 16384 Timesteps
- 8 seconds of data
- Category 0 - Pure Noise
- Category 1 - Signal + Noise
- Frequency of sampling - 2048 Hz

Simulated Dataset - The dataset is simulated using the Python -PYCBC suite. The dataset is divided into:

1. Training data - 24576 (Category 1) & 8192 (Category 0) samples
2. Validation data - 32 (Category 1) & 32 (Category 0) samples
3. Testing data - 8192 (Category 1) & 2048 (Category 0) samples

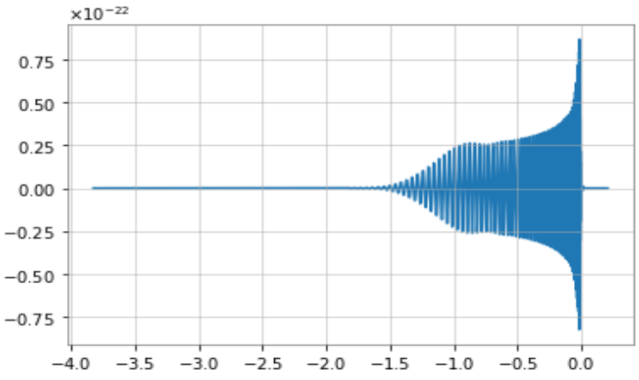


Figure 5.5 Sample Pure Wave of two Black Holes Merging

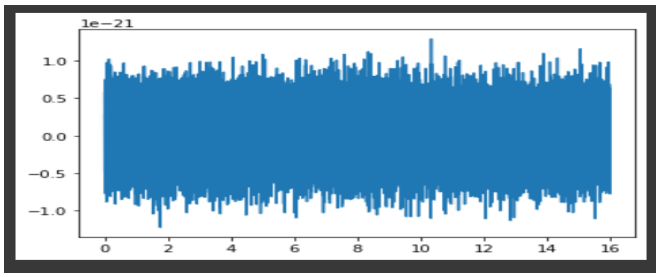
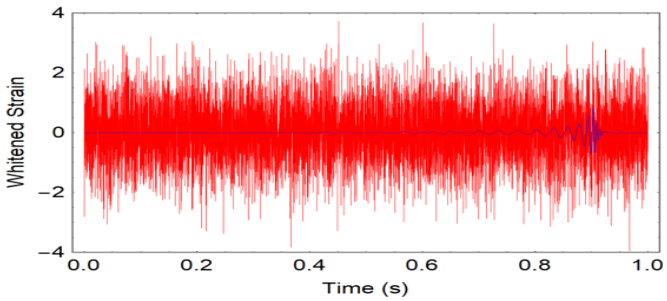


Figure 5.6 Pure Noise wave of 16 secs

Figure 5.7 Ideal Strain



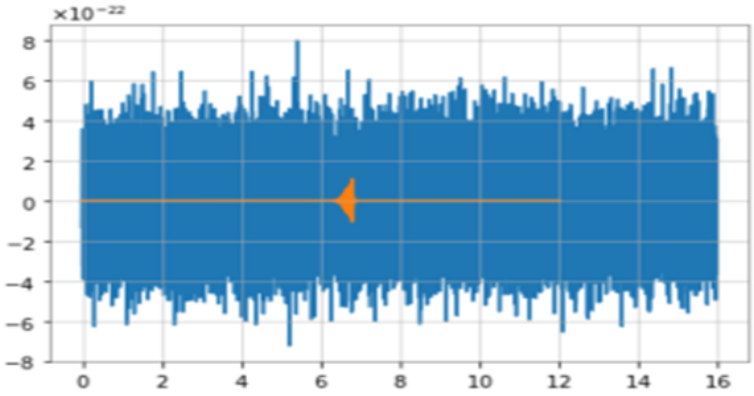
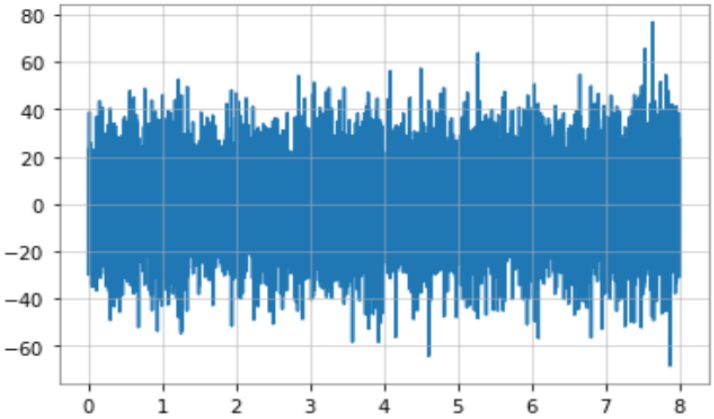


Figure 5.8 Pure wave of 8 sec in noise of 16 sec

Figure 5.9 Final Whitened Signal + Noise wave



Real-Time LIGO, VIRGO Data -

Since there are many fewer observations detected by LIGO and VIRGO detectors hence real-time data can not be used for training and validation, that is why the need arose for the simulated dataset. However, the trained network can be tested for Real-time captured data by the detectors.

Model Results on real-time LIGO data

- Merger - GW151012 taken from L1 i.e. Livingston Detector
- Visualization of data -

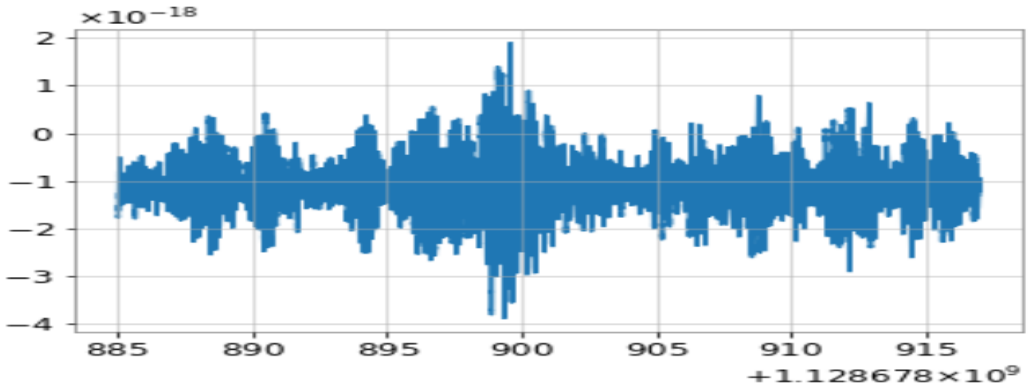


Figure 5.10 Real Time Gravitational Wave Event (151012)

- Processed data as per Model Input: -

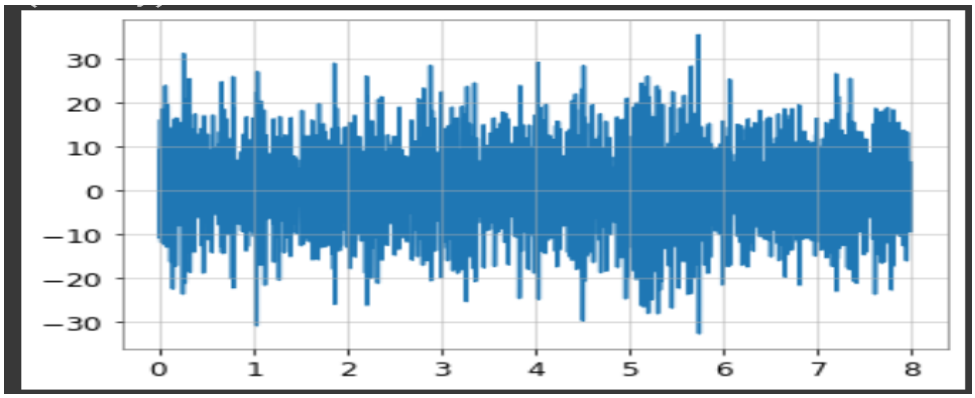


Figure 5.11 Preprocessed Whitened Real Time Gravitational Wave Event (151012)

5.2.2 Denoising:

Input data specifications: -

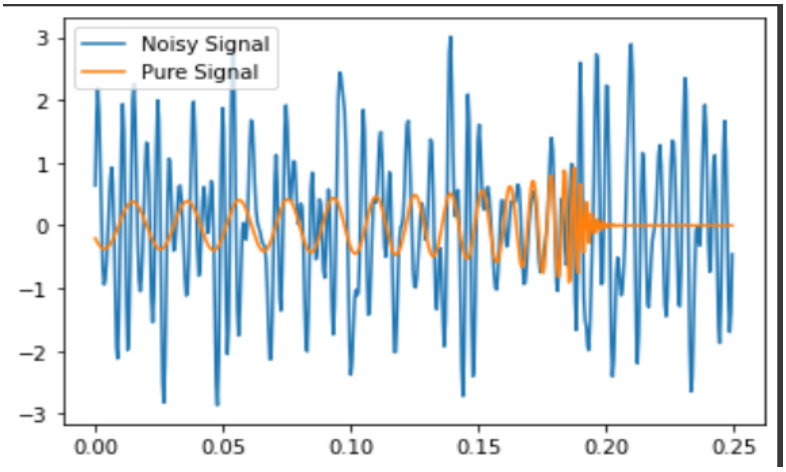
- 512 Timesteps
- 0.25 seconds length of data
- Output will be a reconstructed signal of same dimensions
- Frequency of sampling - 2048 Hz
- 8000 (Signal + Noise) Waves for training.

Simulated Dataset - The dataset is simulated using the Python -PYCBC suite. The dataset is divided into:

1. Training data - 8000 samples
2. Testing data - 3400 samples

Input Data

Figure 5.12 Preprocessed Whitened Simulated Data Sample



5.3 Algorithm:

5.3.1 Classification:

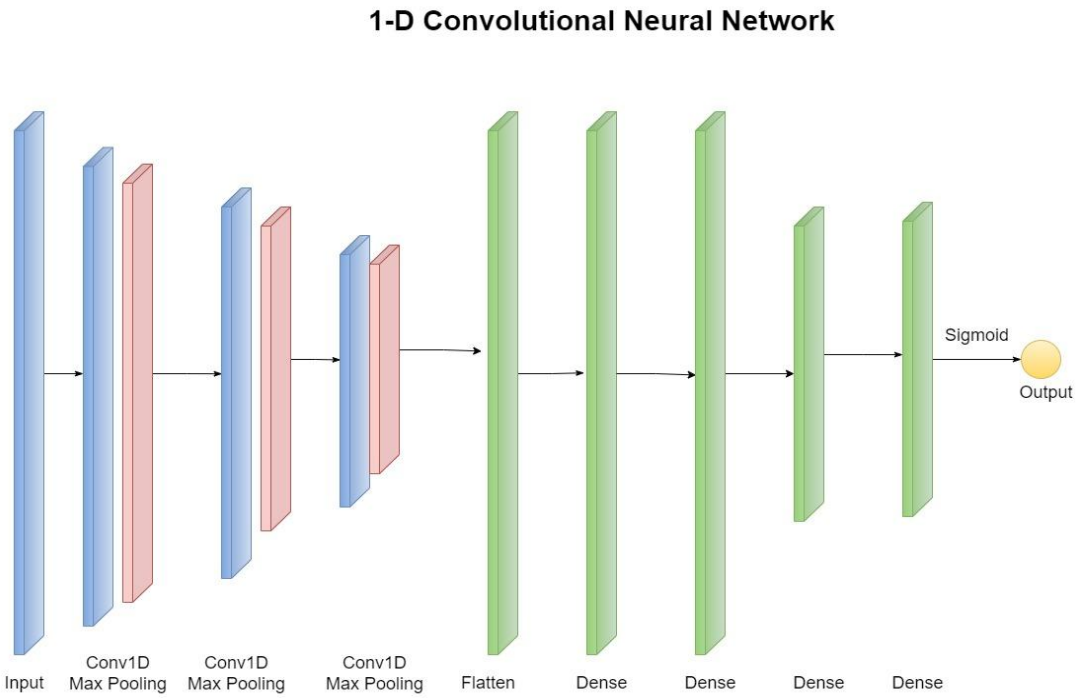


Figure 5.13 Classification - 1D Convolutional Neural Network Model

- The first step is the collection of the dataset. Since the real time data captured by LIGO detectors are very less, hence they aren't enough for training and validation of the model. Hence we used the PYCBC suite in Python for simulation of Binary Black Hole Merger signals as well as Uniform Additive white Gaussian Noise.

- We have created the dataset in the ratio of 3:1 i.e. 3 parts of Noisy Binary Black Hole Merger signals and 1 part of Pure Noise Signals for all the three operations of Training, Validation and Testing.

- The second step involves a number of pre-processing steps which needs to be carried out in any Deep Learning Project. These pre-processing steps are applied while generating simulated data itself. Hence no preprocessing is needed while inputting to the Neural Network. The next step was to generate Pure Noise Signals. After which batches with samples of pure noise waves and samples of BBH mergers waves were created for training, validation and testing.

- Once the preprocessing is done, the data set is trained, validated and tested on the model. The training data set makes sure that the machine recognizes patterns in the data and validation ensures better accuracy and efficiency of the algorithm used to train the machine, and the test dataset is used to see how well the machine can predict new answers based on its training.

- The classifier has been built from scratch using TensorFlow Keras 1D Convolution , Flatten and Dense Layers. Depending on how many and what layers are required, the layers are added with their required shape respectively.

- This is followed by training the model. To train the network model, the waves of BBH Mergers and Pure Noise dataset are generated.

- Once the training process is over the network learns the features of classifying BBH Merger Gravitational waves and Pure Noise signals.

- The training is followed by testing the model. A number of BBH Merger and Noise waves are tested to check the validation of the accuracy. Once it's done, the actual real time gravitational wave captured from the LIGO detector can be used to check whether the wave is a BBH Merger Gravitational Wave or Noise.

5.3.2 Denoising:

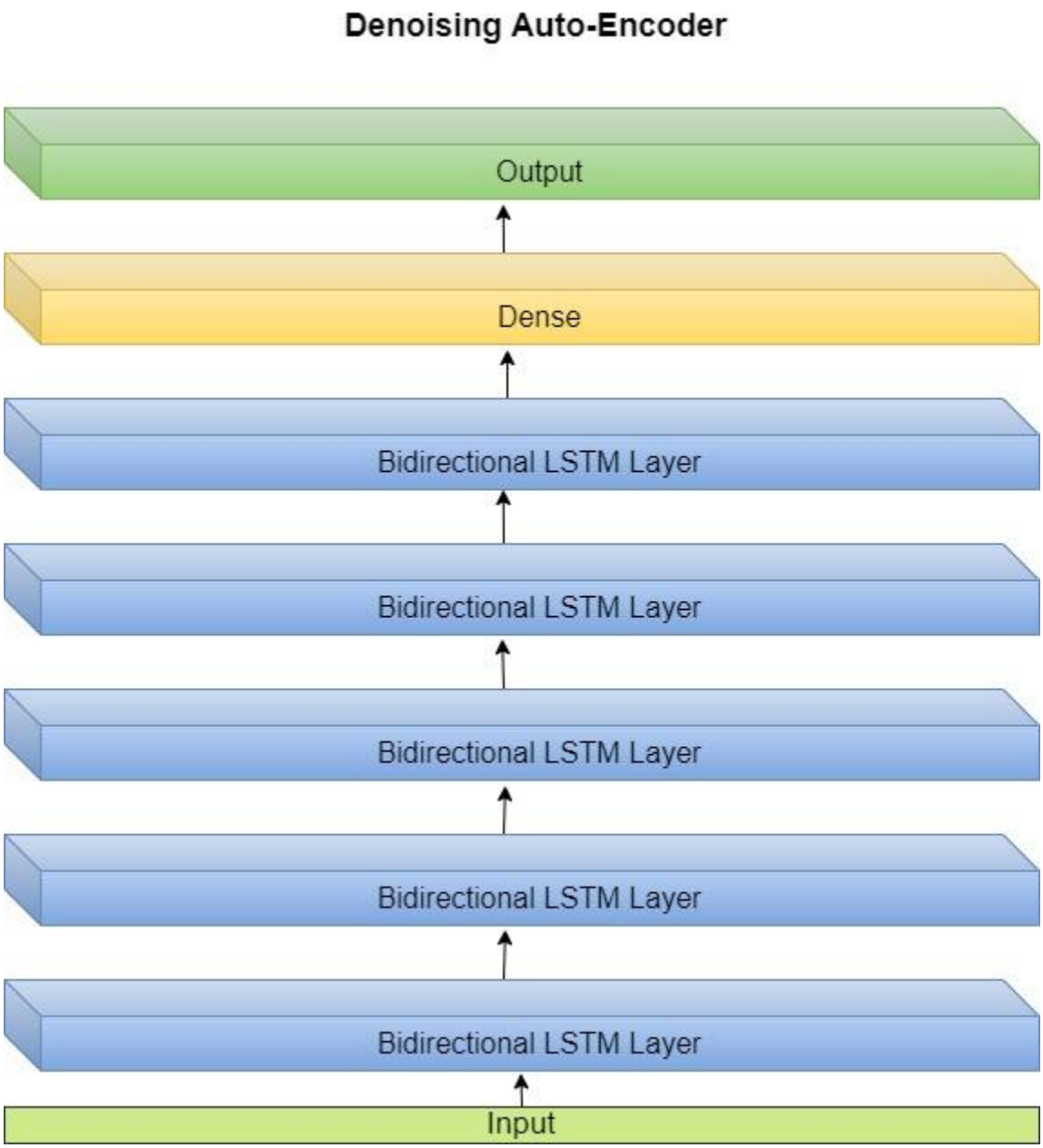


Figure 5.14 Denoising Autoencoder- Bi-Directional LSTM Model

- The first step is the collection of the dataset. We used the PYCBC suite in Python for simulation of Binary Black Hole Merger signals as well as Additive white Gaussian Noise.

- We have created the dataset in batches as per Scaled SNR as follows: [17,16,15,14], [13,12,11,10], [9,8,7,6], [5,4,3,2]. Each batch has 2000 samples with Uniform Distribution of the SNR range. The batches are created for training in accordance with the concept of curriculum learning.

- The preprocessing steps of parameterization, shifting, sampling, noise addition and SNR scaling are the same as classification. The final signal is of 0.25s and 512 samples consisting of scaled BBH Merger signal by a factor of 10^{22} and whitened noise.

- After preprocessing and sample generation, the dataset is trained, validated and tested on the model. Using the training data the model recognizes the pattern in the noisy signal, by comparing it with the provided pure signal of the same noisy signal. The validation dataset is used to check the efficiency and performance of the model. The model is then tested on the testing data to make sure it adapts itself to unseen data and denoises them too with high accuracy.

- The denoising model has been built on the concept of denoising autoencoders. It is built from scratch using TensorFlow Keras Bi-directional LSTM and Dense Layers. Depending on the size of input data and performance, the layers are added with their required shape respectively.

5.4 Challenges Faced

1. The first major challenge was the availability of the desired dataset required for classification. Later on, alternatives such as the PYCBC suite were found using which we could simulate a dataset for training and testing in Python.

2. The second challenge was related to 1D signal preprocessing techniques. The signal had to be tuned as per the signals detected in real time. The appropriate amount of noise had to be added such that the signal doesn't lose its idealistic nature but still possesses the characteristics of a real time signal.

3. We had to implement multiple models for finding the best model which will breach the 99% training accuracy mark. This process was time-consuming as it involved the adjustment of hyperparameters.

4. The next challenge was reducing the computational time during training the classification model. This time was reduced from 12 mins per epoch to 40s per epoch by optimizing the code for GPU usage.

5. The size of the data during classification was also a challenge. The computational complexity increased because of the 16384 timesteps in each sample.

6. The challenge of availability of data for denoising was solved by using PyCBC.

7. The length of data for denoising was a hurdle in getting the desired results. The 8s of data used for classification had a lot of 0s in the pure part of the signal. However, it was tampering with the MSE in the denoising model. Hence the only option was to reduce the length of data which, after hit and trial, was brought down to 0.25s from 8s.

8. The denoising model had unchanging MSE initially when the entire dataset was inputted altogether. However, this problem was solved by using the concept of curriculum learning during training.

CHAPTER 6: SOFTWARE DEVELOPMENT

6.1 Programming Software

6.1.1 Jupyter Notebook: We have tested and implemented our project code in Jupyter Notebook. It is a free, open-source, interactive web tool known as a computational notebook, which users can use to combine software code, computational output, explanatory text and multimedia resources etc everything in a single document. The Jupyter notebook has two components. Users input programming code or text in rectangular cells in a front-end web page. The browser then passes that code to a back-end 'kernel', which runs the code and returns the results. Notebooks can also run in the cloud.

Jupyter has gained traction across many fields as an open-source environment that is compatible with numerous programming languages. The name Jupyter is a reference to the three core languages supported by the project (Julia, Python, and R), but kernels are available that make Jupyter compatible with tens of languages, including Ruby, PHP, Javascript, SQL, and Node.js. It may not make sense to implement projects in all of these languages using Jupyter notebooks (e.g. Omeka won't let you install a plugin written as a Jupyter notebook), but the Jupyter environment can still be valuable for documenting code, teaching programming languages, and providing students with a space where they can easily experiment with provided examples. Why use a Jupyter Notebook?

Advantages of Jupyter Notebook-

- All in one place: As you know, Jupyter Notebook is an open-source web-based interactive environment that combines code, text, images, videos, mathematical equations, plots, maps, graphical user interface and widgets to a single document.
- Easy to share: Jupyter Notebooks are saved in the structured text files (JSON format), which makes them easily shareable.
- Easy to convert: Jupyter Notebook allows users to convert the notebooks into other formats such as HTML and PDF. It also uses online tools and nbviewer which allows you to render a publicly available notebook in the browser directly.

6.1.2 Google Colab: Over many years, Google developed an AI framework called TensorFlow and a development tool called Colaboratory. Google made Colaboratory free for public use. Colaboratory is now known as Google Colab. Colab supports GPU and it is totally free. The reasons for making it free for the public could be to make its software a standard in the academics for teaching machine learning and data science. The introduction of Colab has eased the learning and development of machine learning applications.

Colab is a free Jupyter notebook environment that runs entirely in the cloud. Most importantly, it does not require a setup and the notebooks that you create can be simultaneously edited by your team members - just the way you edit documents in Google Docs. Colab supports many popular machine learning libraries which can be easily loaded in your notebook. What Colab Offers You?

As a programmer, you can perform the following using Google Colab:

- Write and execute code in Python
- Document your code that supports mathematical equations
- Create/Upload/Share notebooks
- Import/Save notebooks from/to Google Drive
- Import/Publish notebooks from GitHub
- Import external datasets e.g. from Kaggle
- Integrate PyTorch, TensorFlow, Keras, OpenCV
- Free Cloud service with free GPU

6.1.3 Kaggle: Kaggle, a subsidiary of Google LLC, is an online community of data scientists and machine learning practitioners. Kaggle allows users to find and publish data sets, explore and build models in a web-based data-science environment, work with other data scientists and machine learning engineers, and enter competitions to solve data science challenges.

6.2 Programming language:

We have done the coding and implementation of our project in Python. We have used Python because Python is user friendly and the syntax in python is very simple and easy. Python is a general-purpose high-level programming language designed to be easy to read and simple to implement. It is an interpreted and object-oriented programming language. It is open source which means it is free to use by everyone. It has various open source libraries which are available free of cost. Python is meant to be an easily readable language. Python allows programmers to define their own types using classes. Python works on different platforms (Windows, Mac, Linux, Raspberry Pi). Python has syntax that allows developers to write programs with fewer lines than some other programming languages. Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick. Python can be treated in a procedural way, an object-oriented way or a functional way.

6.3 Python Libraries Used

1) **Numpy** : NumPy a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. NumPy is open-source software and has many contributors. NumPy is the most basic yet powerful package for scientific computing and data manipulation in Python. It is an open source library available in Python. It helps to do mathematical and scientific operations and is used extensively in data science. It allows us to work with multidimensional arrays and matrices. NumPy is the fundamental package for scientific computing with Python. It contains among other things:

    - a powerful N-dimensional array object
    - sophisticated (broadcasting) functions
    - tools for integrating C/C++ and Fortran code
    - useful linear algebra, Fourier transform, and random number capabilities

    Besides its obvious scientific uses, NumPy can also be used as an efficient

multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

2) **Matplotlib:** Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK+. Pyplot is a Matplotlib module which provides a MATLAB-like interface. Matplotlib is designed to be as usable as MATLAB, with the ability to use Python, and the advantage of being free and open-source. Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

3) **PyCBC:** PyCBC is a software package used to explore astrophysical sources of gravitational waves. It contains algorithms to analyze gravitational-wave data, detect coalescing compact binaries, and make bayesian inferences from gravitational-wave data. PyCBC was used in the first direct detection of gravitational waves and is used in flagship analyses of LIGO and Virgo data. PyCBC is collaboratively developed by the community and is led by a team of GW astronomers with the aim to build accessible tools for gravitational-wave data analysis. PyCBC was used in the first direct detection of gravitational waves (GW150914) by LIGO.

4) **TensorFlow:** TensorFLow is an end-to-end open-source deep learning framework developed by Google and released in 2015. It is known for documentation and training support, scalable production and deployment options, multiple abstraction levels, and support for different platforms, such as Android.TensorFlow is a symbolic math library used for neural networks and is best suited for dataflow programming across a range of tasks. It offers multiple abstraction levels for building and training models. A promising and fast-growing entry in the world of deep learning, TensorFlow offers a flexible, comprehensive ecosystem of community resources, libraries, and tools that facilitate building and deploying machine learning apps.

5) **Keras:** Keras is an effective high-level neural network Application Programming Interface (API) written in Python. This open-source neural network library is designed to

provide fast experimentation with deep neural networks, and it can run on top of TensorFlow.

6.4 Deep Learning Libraries:

1) **Sequential:** Sequential is the easiest way to build a model in Keras. It allows you to build a model layer by layer. Each layer has weights that correspond to the layer that follows it. We use the 'add()' function to add layers to our model. We will add two layers and an output layer. Models in Keras can come in two forms – Sequential and via the Functional API. For most deep learning networks that you build, the Sequential model is likely what you will use. It allows you to easily stack sequential layers (and even recurrent layers) of the network in order from input to output.

2) **Dense:** A fully connected layer also known as the dense layer, in which the results of the convolutional layers are fed through one or more neural layers to generate a prediction.

3) **Flatten:** In between the convolutional layer and the fully connected layer, there is a 'Flatten' layer. Flattening transforms a two-dimensional matrix of features into a vector that can be fed into a fully connected neural network classifier. Flatten is the function that converts the pooled feature map to a single column that is passed to the fully connected layer. Dense adds the fully connected layer to the neural network.

4) **1D Convolutional Layer:** A 1-D convolutional layer applies sliding convolutional filters to 1-D input. The layer convolves the input by moving the filters along the input and computing the dot product of the weights and the input, then adding a bias term. The dimension that the layer convolves over depends on the layer input. For time series and vector sequence input (data with three dimensions corresponding to the channels, observations, and time steps), the layer convolves over the time dimension.

5) **Max Pooling:** Pooling is a feature commonly imbibed into CNN architectures. The main idea behind a pooling layer is to "accumulate" features from maps generated by convolving a filter over the data. Formally, its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network. Max pooling is done in part to help over-fitting by providing an abstracted

form of the representation. It reduces the computational cost by reducing the number of parameters to learn and provides basic translation invariance to the internal representation. Max pooling is done by applying a max filter to (usually) non-overlapping subregions of the initial representation.

6) **Dropout:** Overfitting in the model occurs when it shows more accuracy on the training data but less accuracy on the test data or unseen data. In the dropout technique, some of the neurons in hidden or visible layers are dropped or omitted randomly. The experiments show that this dropout technique regularizes the neural network model to produce a robust model which does not overfit.

7) **Activation:** In a neural network, the activation function is responsible for transforming the summed weighted input from the node into the activation of the node or output for that input.

8) **Bi-Directional LSTM:** Bidirectional long-short term memory(bi-lstm) is the process of making any neural network to have the sequence information in both directions backwards (future to past) or forward(past to future). In bidirectional, our input flows in two directions, making a bi-lstm different from the regular LSTM. With the regular LSTM, we can make input flow in one direction, either backwards or forward. However, in bi-directional, we can make the input flow in both directions to preserve the future and the past information.

CHAPTER 7: RESULTS AND DISCUSSIONS

7.1 Classification:

The Neural Network model created here is a Dense 1D Convolutional Neural Network model, which connects each layer to every other layer in a feed-forward fashion.

Model Training and Validation: It is a problem of Binary Classification. During the training of the model, it is observed that at each epoch, the accuracy increases and gives better results. Since our project does not involve a very high number of datasets, only 14 epochs have been used for the training. The time taken for training is observed to be around 11 minutes. The accuracy of the trained model in the 14th epoch is seen to be as 99.2% and the validation accuracy is seen to be as 96.88% which seems to be completely reliable for our project. The GPU used for training is the TESLA K80. The total trainable parameters were 1.07 million. The loss metric was 'Binary cross entropy'. The accuracy metric is 'Classification accuracy'. We have used the 'ADAM' optimiser with a learning rate of 0.002.

EPOCH	TRAIN ACCURACY	TRAIN LOSS	VAL ACCURACY	TIME
1	0.7384	0.6619	0.7656	00:41
2	0.8734	0.2515	0.9375	00:58
3	0.9371	0.1580	0.9688	01:04
4	0.9557	0.1156	0.9688	00:57
5	0.9678	0.0856	0.9531	00:52
6	0.9743	0.0681	0.8906	00:40
7	0.9796	0.0563	0.9688	00:38
8	0.9819	0.0479	0.9688	00:50
9	0.9837	0.0424	0.9688	00:40
10	0.9872	0.0321	0.9844	00:38

11	0.9879	0.0296	0.9688	00:36
12	0.9911	0.0238	0.9688	00:37
13	0.9917	0.0211	0.9688	00:36
14	0.9919	0.0196	0.9688	00:37

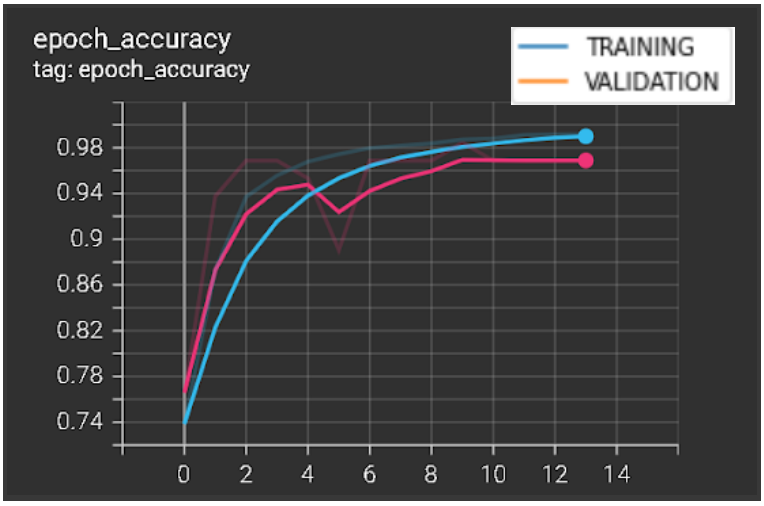


Figure 7.1 Accuracy vs Epochs

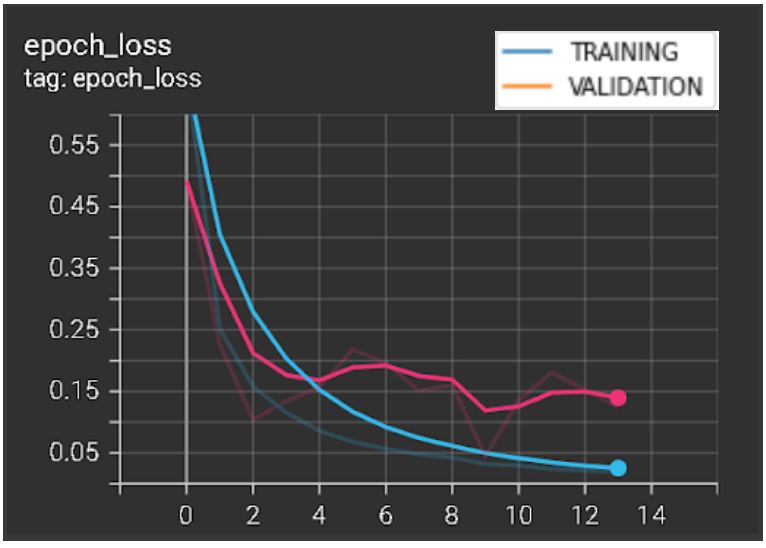


Figure 7.2 Loss vs Epochs

Classification Model Architecture:

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv1d_3 (Conv1D)            (None, None, 8)           72
 max_pooling1d_3 (MaxPooling  (None, None, 8)           0
 1D)
 conv1d_4 (Conv1D)            (None, None, 16)          1040
 max_pooling1d_4 (MaxPooling  (None, None, 16)          0
 1D)
 conv1d_5 (Conv1D)            (None, None, 32)          4128
 max_pooling1d_5 (MaxPooling  (None, None, 32)          0
 1D)
 flatten_1 (Flatten)          (None, None)              0
 dense_5 (Dense)              (None, 128)               1036416
 dropout_4 (Dropout)          (None, 128)               0
 dense_6 (Dense)              (None, 128)               16512
 dropout_5 (Dropout)          (None, 128)               0
 dense_7 (Dense)              (None, 64)                8256
 dropout_6 (Dropout)          (None, 64)                0
 dense_8 (Dense)              (None, 64)                4160
 dropout_7 (Dropout)          (None, 64)                0
 dense_9 (Dense)              (None, 1)                 65
=================================================================
Total params: 1,070,649
Trainable params: 1,070,649
Non-trainable params: 0
_________________________________________________________________
```

Testing: The trained model was tested on the test data with the test loss of 0.13428 and testing accuracy of 97.50%.

The model was also tested on real time LIGO data(not included in the training or testing set). This is a generalized sample unseen by the model.

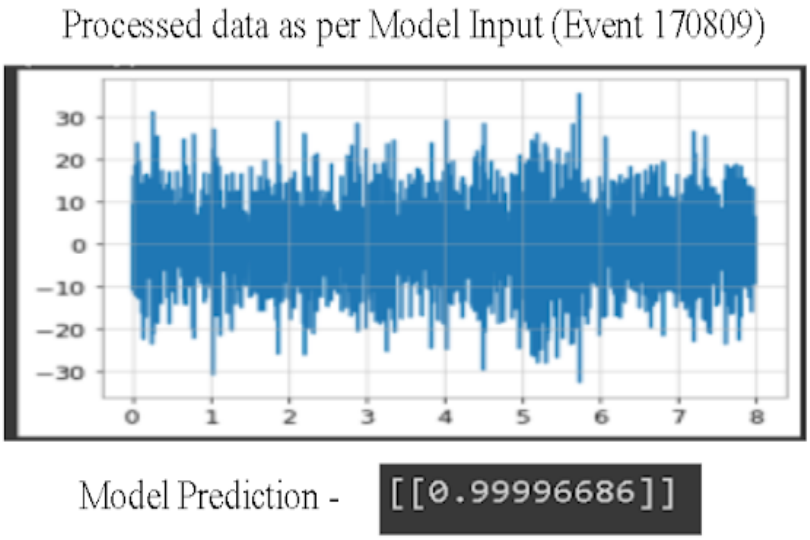


Figure 7.3 Model Prediction on Real Time Gravitational wave data

Thus the model predicted that the wave was 99% a BBH merger wave. Hence it signifies the model has generalized and there is no overfitting.

7.2 Denoising:

The Neural Network model created here is a Dense 1D Bi-Directional LSTM Neural Network model, which works on the principle of Recurrent Neural Network where the output neuron is connected to the input neuron in the form of a feedback.

Model Training and Validation: This is essentially a dimensionality reduction problem. We have used the concept of Curriculum Learning. We created batches of signals with particular SNRs as follows:

1. Batch 1 - [17,16,15,14]
2. Batch 2 - [13,12,11,10]
3. Batch 3 - [9,8,7,6]
4. Batch 4 - [5,4,3,2]

Each batch consists of 2000 samples. The model is trained on each batch one by one by decreasing the signal's SNR. Since the number of data points was high, we trained the

model for 60 epochs in the batch sequence 1,2,3,4 then again 3, 4. The time taken for training is observed to be around 13 minutes. The GPU used for training is the TESLA P100. The total trainable parameters were 1.07 million. The loss metric was 'Mean Square Error(MSE)'. The accuracy metric is the loss metric itself, since we have found the error between the noisy and pure wave. We have used the 'ADAM' optimiser.

The MSE at the end of 60 epochs was 0.0014.

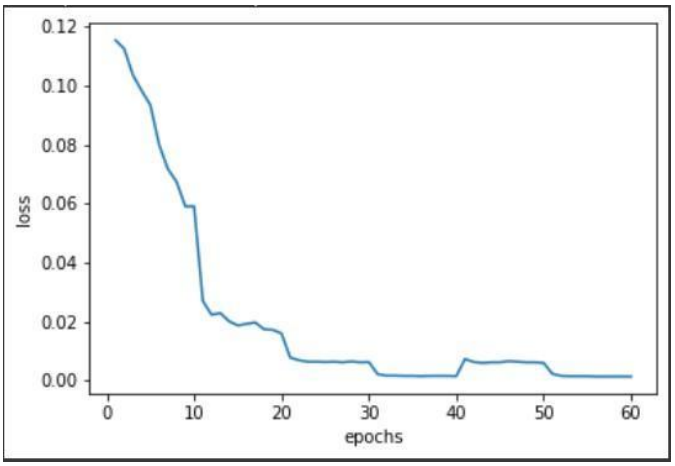


Figure 7.4 MSE vs Epochs

Denoising Autoencoder Architecture:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 bidirectional (Bidirectiona  (None, None, 512)        528384
 l)
 dropout (Dropout)           (None, None, 512)         0
 bidirectional_1 (Bidirectio  (None, None, 32)         67712
 nal)
 dropout_1 (Dropout)         (None, None, 32)          0
 bidirectional_2 (Bidirectio  (None, None, 2)          272
 nal)
 dropout_2 (Dropout)         (None, None, 2)           0
 bidirectional_3 (Bidirectio  (None, None, 32)         2432
 nal)
 dropout_3 (Dropout)         (None, None, 32)          0
 bidirectional_4 (Bidirectio  (None, 512)              591872
 nal)
 dropout_4 (Dropout)         (None, 512)               0
 dense (Dense)               (None, 512)               262656
 activation (Activation)     (None, 512)               0
=================================================================
Total params: 1,453,328
Trainable params: 1,453,328
Non-trainable params: 0
_________________________________________________________________
```

Testing:

The testing dataset had 200 samples of each batch category, comprising 50 samples of each SNR wave. The trained model was tested on each test batch and the results are as follows:

1. Test Batch 1 - 0.07015499472618103
2. Test Batch 2 - 0.0314708910882473
3. Test Batch 3 - 0.008075942285358906
4. Test Batch 4 - 0.001213066279888153

We also tested our model on a new batch with SNR -[0.5,1,1.5,2] which was not seen by the model during training. This testing data was created to check the generalization of the model with the results, Test Batch 5 - 0.0004652967036236077 5.

A mixed data set with SNRs from 0.5 to 17 was created with steps of 0.5, 50 samples of each SNR wave. The MSE was calculated for each SNR and the graph was plotted.

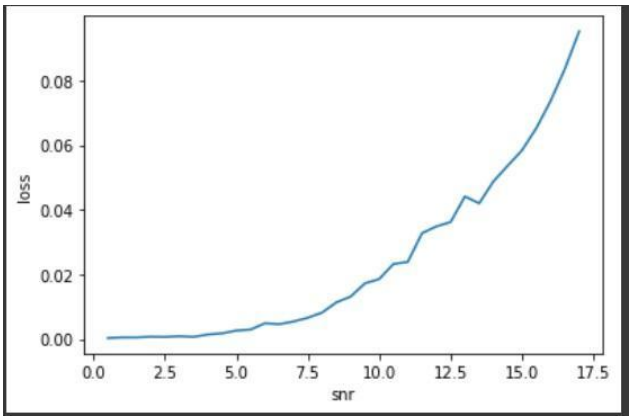


Figure 7.5 MSE Loss vs SNR

Model testing on simulated samples (not included in the training or testing set):

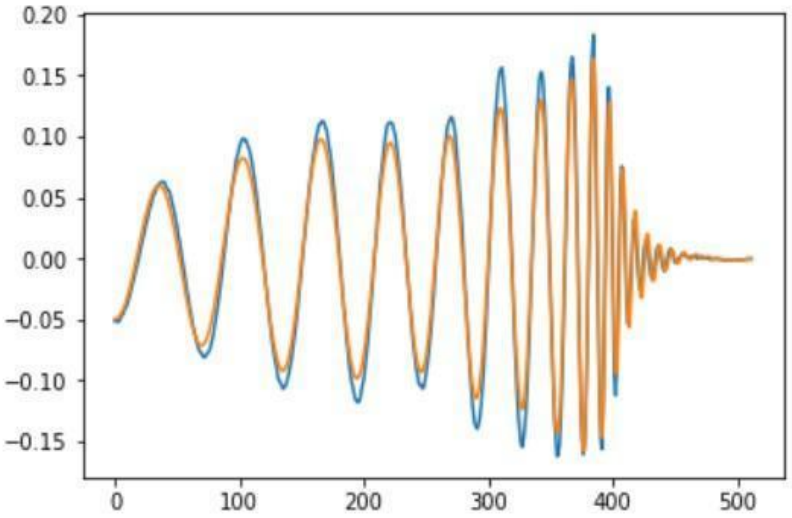


Figure 7.6 Result of Denoising Autoencoder on data samples

Yellow: Reconstructed Signal Blue: Pure Signal

CHAPTER 8: CONCLUSION AND FUTURE WORK

The previous chapter explained the results of our project and discussed its justified analysis. Thus, from the above discussions and detailed analysis of the results, we derive certain conclusions which help us understand the outcome of the objective of our project.

8.1 Conclusion:

1. The project involved preprocessing, classification and denoising of real time gravitational wave signals.
2. After extensive research, the behavior of gravitational waves were analyzed and understood.
3. The project focused on only one source of gravitational waves called the Compact Binary Coalescence which were simulated using Python. After visualization of waves, it can be concluded that the waveform is a function of frequency, masses of merging bodies, spin of bodies and distance between the two merging bodies.
4. The Classification model solved the problem of binary classification and categorized the simulated data in two categories: BBH Merger Signals and Pure Noise.
5. The accuracy achieved by the classification model was better than those obtained in previous research, thus rendering the process efficient.
6. The classification model was also found to be accurate on real time gravitational wave data directly extracted from the LIGO detector.
7. By taking into consideration the GPU type, its processing speed and memory space, it can be concluded that our model requires less computational time and space.
8. The denoising model achieves lesser error as compared to the author and denoises the simulated gravitational waves near to perfection.
9. From the results of the denoising model on unseen data samples, it can be concluded that the model generalizes efficiently to newer samples and does not overfit.
10. It can be concluded that by using curriculum learning the error of the denoising model can be reduced, thus creating a better, efficient model.
11. The denoising model was found to be giving low MSE for low SNR signals and high

MSE for high SNR signals, however the slope of the graph is not too steep. Hence it can be said the model works very efficiently for low SNR signals and moderately high signals, thus it can be implemented for the entire range of SNR without any significant loss of data.

8.2 Future Scope:

1. Number of output classes of the classification model can be increased, which will segregate different types of gravitational waves.
2. By using the concept of Transfer Learning, the model can be used for classification of different noises within a gravitational wave.
3. Implementation of the denoising model for different types of noise e.g. non-stationary noise, non-gaussian noise, glitches, etc.
4. Denoising model architecture can be used for other deep space signals using transfer learning.
5. The denoised signal can be used for parameter estimation to get the parameters responsible for the wave pattern.
6. Hyperparameters can be tuned to achieve better accuracy.
7. The model can be implemented on customized FPGA for faster inference during real time testing.

APPENDIX

Gravitational Wave Pure and Noisy BBH signal Generation

```
import sys
!{sys.executable} -m pip install pycbc ligo-common --no-cache-dir

import numpy as np
import math
import pylab

import matplotlib.pyplot as plt
import random
import pycbc
from pycbc import distributions
from pycbc.waveform import get_td_waveform
from pycbc.detector import Detector
import pycbc.coordinates as co
from pycbc.psd import welch, interpolate
from pycbc.psd import interpolate, inverse_spectrum_truncation
from pycbc.noise.gaussian import noise_from_psd
from pycbc.noise.gaussian import frequency_noise_from_psd
from pycbc.filter import matched_filter

det_l1 = Detector('L1')
apx = 'IMRPhenomD'
N=2048*16  #N is number of samples, N=length/delta_t
fs=2048 #fs is sampling frequnecy
length=16 #duration of segment
delta_f=1.0/16
f_samples = 16385
f_lower=40
delta_t=1.0/2048

from pycbc.psd.analytical import AdVDesignSensitivityP1200087

def get_psd(f_samples, delta_f, low_freq_cutoff):
    psd=AdVDesignSensitivityP1200087(f_samples, delta_f,
low_freq_cutoff)
    return psd

from pycbc.noise.gaussian import frequency_noise_from_psd

def get_noise(psd, seed=None):
    noise=frequency_noise_from_psd(psd, seed=seed)
    noise_time = noise.to_timeseries()
    return noise_time

def add_noise_signal(noise, signal):
    length_signal = len(signal)
    signal_plus_noise=noise
```

```
signal_plus_noise[0:length_signal]=np.add(noise[0:length_signal],
signal)
    return signal_plus_noise


from pycbc.psd import welch, interpolate

def get_whiten(signal_plus_noise):
    signal_freq_series=signal_plus_noise.to_frequencyseries()
    numerator = signal_freq_series
    psd_to_whiten = interpolate(welch(signal_plus_noise), 1.0 /
signal_plus_noise.duration)
    denominator=np.sqrt(psd_to_whiten)
    whiten_freq = (numerator / denominator)
    whiten=whiten_freq.to_timeseries().highpass_fir(30.,
512).lowpass_fir(300.0, 512)
    return whiten

def get_8s(whiten, signal_peak_index=None):
    whiten.start_time = 0
    cropped = whiten.time_slice(0,8)
    return cropped

psd=get_psd(f_samples, delta_f, f_lower)

def DISTRIBUTIONS(low, high, samples):
    var_dist = distributions.Uniform(var = (low, high))
    return var_dist.rvs(size = samples)

def SPIN_DISTRIBUTIONS(samples):
    theta_low = 0.
    theta_high = 1.
    phi_low = 0.
    phi_high = 2.
    uniform_solid_angle_distribution =
distributions.UniformSolidAngle(polar_bounds=(theta_low,theta_high),

azimuthal_bounds=(phi_low,phi_high))
    solid_angle_samples =
uniform_solid_angle_distribution.rvs(size=samples)
    spin_mag = np.ndarray(shape=(samples), dtype=float)
    for i in range(0,samples):
        spin_mag[i] = 1.
    spinx, spiny, spinz =
co.spherical_to_cartesian(spin_mag,solid_angle_samples['phi'],solid_
angle_samples['theta'])
    return spinz

def get_params(samples):
    mass1_samples = DISTRIBUTIONS(10, 80, samples)
    mass2_samples = DISTRIBUTIONS(10, 80, samples)
    right_ascension_samples  = DISTRIBUTIONS(0 , 2*math.pi, samples)
```

```
    polarization_samples = DISTRIBUTIONS(0 , 2*math.pi, samples)
    declination_samples = DISTRIBUTIONS((-math.pi/2)+0.0001,
(math.pi/2)-0.0001, samples)
    spinz1 = SPIN_DISTRIBUTIONS(samples)
    spinz2 = SPIN_DISTRIBUTIONS(samples)
    snr_req = DISTRIBUTIONS(2, 17, samples)
    DIST = DISTRIBUTIONS(2500, 3000, samples)
    return mass1_samples, mass2_samples, right_ascension_samples,
polarization_samples, declination_samples, spinz1, spinz2, snr_req,
DIST

Training_samples = np.zeros((24576, 8192*2))
def DATA_GENERATION(samples):

  mass1_samples, mass2_samples, right_ascension_samples,
polarization_samples, declination_samples, spinz1, spinz2, snr_req,
DIST = get_params(samples)
  for i in range(0,samples):
        seed =  random.randint(1, 256)
        # NOTE: Inclination runs from 0 to pi, with poles at 0 and
pi
        #       coa_phase runs from 0 to 2 pi.
        hp, hc = get_td_waveform(approximant=apx,
                                 mass1=mass1_samples[i][0],
                                 mass2=mass2_samples[i][0],
                                 spin1z=spinz1[i],
                                 spin2z=spinz2[i],
                                 delta_t=delta_t,
                                 distance = DIST[i][0],
                                 f_lower=30)

        signal_l1 = det_l1.project_wave(hp, hc,
right_ascension_samples[i][0], declination_samples[i][0],
polarization_samples[i][0])
        signal_l1.append_zeros(10*2048)
        signal_l1 = signal_l1.cyclic_time_shift(5)
        signal_l1.start_time = 0

        noise=get_noise(psd)
        signal_l1 = get_8s(signal_l1)
        final = add_noise_signal(noise, signal_l1)

        whiten = get_whiten (final)

        data = get_8s(whiten)
        Training_samples[i] = data

DATA_GENERATION(24576)


name = '/content/gdrive/My Drive/Positive_Training_DATA'
np.save(name, Training_samples)
```

```
Negative_Training_samples = np.zeros((8192, 8192*2))


def NOISE_GENERATOR(samples):
  for i in range(samples):
    seed =  random.randint(1, 1000)
    noise=get_noise(psd, seed)
    # pylab.plot(noise.sample_times, noise)
    whiten = get_whiten (noise)
    noise_signal = get_8s(whiten)
    # pylab.plot(noise_signal.sample_times, noise_signal)
    # print(len(noise_signal))
    # print(noise_signal[0])
    Negative_Training_samples[i] = noise_signal

NOISE_GENERATOR(8192)
name = '/content/gdrive/My
Drive/Classification_Negative_Training_Samples'
np.save(name, Negative_Training_samples)
```

Classification Model

```
from google.colab import drive
drive.mount('/content/gdrive',force_remount=True)



import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Dense, Flatten, MaxPooling1D,
GlobalAveragePooling1D
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, Dropout
from tensorflow.keras.optimizers import Adam, RMSprop
from tensorflow.keras.losses import BinaryCrossentropy
from tensorflow.keras.metrics import binary_accuracy, AUC
from sklearn.utils import shuffle

from tensorflow.keras.callbacks import TensorBoard
import time

NAME ="BBH_ReClassification_{}".format(int(time.time()))
tensorboard = TensorBoard(log_dir
='/content/gdrive/MyDrive/FYPNBs/Re_Classification/logs/{}'.format(N
AME))
name = '/content/gdrive/MyDrive/FYP
NBs/Re_Classification/Training_DATA'

def model_creation():
  n_timesteps, n_features = 16384, 1
  input_shape=(n_timesteps,n_features)
```

```
  model=Sequential()

  model.add(Conv1D(filters=8, kernel_size=8, activation='relu'))
  # model.add(Conv1D(8, kernel_size=8, activation='relu'))
  model.add(MaxPooling1D(pool_size=4))

  model.add(Conv1D(16, kernel_size=8, activation='relu'))
  # model.add(Conv1D(16, kernel_size=8, activation='relu'))
  model.add(MaxPooling1D(pool_size=4))

  model.add(Conv1D(32, kernel_size=8, activation='relu'))
  # model.add(Conv1D(64, kernel_size=8, activation='relu'))
  model.add(MaxPooling1D(pool_size=4))

  # model.add(Conv1D(64, kernel_size=8, activation='relu'))
  # model.add(Conv1D(128, kernel_size=8, activation='relu'))
  # model.add(MaxPooling1D(pool_size=4))

  model.add(Flatten())
  # model.add(GlobalAveragePooling1D(data_format='channels_last'))
  model.add(Dense(128, activation='relu'))
  model.add(Dropout(0.3))

  model.add(Dense(128, activation='relu'))
  model.add(Dropout(0.3))

  model.add(Dense(64, activation='relu'))
  model.add(Dropout(0.3))

  model.add(Dense(64, activation='relu'))
  model.add(Dropout(0.3))

  model.add(Dense(1, activation='sigmoid'))

  model.compile(optimizer= Adam(learning_rate=0.002, beta_1=0.9,
beta_2=0.999, epsilon=1e-08),
  loss='binary_crossentropy',
  metrics=['accuracy'])
  return model

model = model_creation()
index = 0
batch_samples = 512
batches= 64
Y  = np.concatenate((np.ones((384, 1)), np.zeros((128, 1))))
X = np.zeros((512, 8192*2))

def get_data():
  global index
  global batch_samples
  global Y
  index = 0
```

```
  while True:
    file = name +'/'+ str(index)+'.npy'
    X = np.load(file)
    trainY = Y
    trainX = X.reshape((512, 8192*2,1))
    trainX, trainY = shuffle(trainX, trainY, random_state=103)
    if(index>=63):
      index = 0
    yield trainX, trainY
    index+=1

X_val = np.load('/content/gdrive/MyDrive/FYP
NBs/Re_Classification/Validation_DATA.npy')
Y_val = np.concatenate((np.ones((32, 1)), np.zeros((32, 1))))
X_val = X_val.reshape((64, 8192*2,1))
X_val, Y_val = shuffle(X_val, Y_val, random_state=5)

from tensorflow.keras.callbacks import ModelCheckpoint
import time
checkpoint_path = "/content/gdrive/MyDrive/FYP
NBs/Re_Classification/Re_BBH_Classification_2.ckpt"


# Create a callback that saves the model's weights every 5 epochs
cp_callback = ModelCheckpoint(
    filepath=checkpoint_path,
    verbose=0,
    save_weights_only=True,
    save_freq = 1)

def model_training():
    global samples
    global batch_samples
    global model


    model.fit(get_data() , epochs=14 , verbose = 1, steps_per_epoch
= 64 , shuffle= True, use_multiprocessing = True,
callbacks=[cp_callback, tensorboard], validation_data = (X_val,
Y_val))

!nvidia-smi -L

global_start_time = time.time()
model_training()
print('Training duration (min) : ', (time.time() -
global_start_time)/60)

model.summary()
%load_ext tensorboard
%tensorboard --logdir '/content/gdrive/MyDrive/FYP
```

```
NBs/Re_Classification/logs/'
```

Classification Model Testing Code

```
from google.colab import drive
drive.mount('/content/gdrive',force_remount=True)
!ls -lt '/content/gdrive/My Drive/'
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Dense, Flatten, MaxPooling1D
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, Dropout
from tensorflow.keras.optimizers import Adam, RMSprop
from tensorflow.keras.losses import BinaryCrossentropy
from tensorflow.keras.metrics import binary_accuracy, AUC
from sklearn.utils import shuffle
from keras.models import load_model

postestsamples = 12288
negtestsamples = 4096

test_samples = 16384
test_X=np.load("/content/gdrive/MyDrive/FYP
NBs/Re_Classification/Classification_Test_DATA.npy")

test_y = np.zeros((test_samples,1))
test_y=np.concatenate((np.ones((postestsamples, 1)),
np.zeros((negtestsamples, 1))))

test_X = test_X.reshape((test_samples, 8192*2, 1))
test_X, test_y = shuffle(test_X, test_y, random_state=127)

model = model_creation()
model_2.load_weights("/content/gdrive/MyDrive/FYP
NBs/Re_Classification/Re_BBH_Classification_2.ckpt")
score = model_2.evaluate(test_X, test_y, verbose = 0)

print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Real Time Data Testing of Classification Model

```
%matplotlib inline

import pylab
from pycbc.catalog import Merger
from pycbc.filter import resample_to_delta_t, highpass, lowpass

# As an example we use the GW150914 data
merger = Merger("GW151012")

# Get the data from the Livingston detector
strain = merger.strain('L1')
# print(strain.shape)
# strain.start_time = 0
pylab.plot(strain.sample_times, strain)

strain_10s = strain.time_slice(merger.time-6.0, merger.time+6.0)
strain_10s.start_time = 0
pylab.plot(strain_10s.sample_times, strain_10s)
strain_fil = highpass(strain_10s, 25.0)
pylab.plot(strain_fil.sample_times, strain_fil)
strain_res = resample_to_delta_t(strain_fil, 1.0/2048)
pylab.plot(strain_res.sample_times, strain_res)
conditioned = strain_res.crop(1, 1)
conditioned.start_time = 0
pylab.plot(conditioned.sample_times, conditioned)
print(conditioned.shape)

from pycbc.psd import welch, interpolate
def get_whiten(signal_plus_noise):
    signal_freq_series=signal_plus_noise.to_frequencyseries()
    numerator = signal_freq_series
    psd_to_whiten = interpolate(welch(signal_plus_noise), 1.0 /
signal_plus_noise.duration)
    denominator=np.sqrt(psd_to_whiten)
    whiten_freq = (numerator / denominator)
    whiten=whiten_freq.to_timeseries().highpass_fir(30.,
512).lowpass_fir(100.0, 512)
    return whiten

whiten_real = get_whiten(conditioned)
pylab.plot(whiten_real.sample_times, whiten_real)
print(whiten_real.shape)

def get_8s(whiten, signal_peak_index=None):
    whiten.start_time = 0
    cropped = whiten.time_slice(0,8)
    return cropped

real_sig = get_8s(whiten_real)
pylab.plot(real_sig.sample_times, real_sig)
```

```
print(real_sig.shape)

model = model_creation()
model.load_weights("/content/gdrive/MyDrive/FYP
NBs/Re_Classification/Re_BBH_Classification_2.ckpt")
# model =
load_model("/content/gdrive/MyDrive/BBH_Classification_Model_5.h5")
# model.load_weights("/content/gdrive/My
Drive/BBH_Classification_5.ckpt")
sample =np.zeros((1,8192*2))
sample[0] = real_sig
sample = sample.reshape(1, 16384,1)
prediction= model.predict(sample)
print(prediction)
```

Data Generation Code for Denoising Model

```
import numpy as np
import math
import pylab
%matplotlib inline
import matplotlib.pyplot as plt
import random
import pycbc
from pycbc import distributions
from pycbc.waveform import get_td_waveform
from pycbc.detector import Detector
import pycbc.coordinates as co
from pycbc.psd import welch, interpolate
from pycbc.psd import interpolate, inverse_spectrum_truncation
from pycbc.noise.gaussian import noise_from_psd
from pycbc.noise.gaussian import frequency_noise_from_psd
from pycbc.filter import matched_filter

snr_REQ = []
for i in range(5,175, 5):
  for j in range(0,50):
    f = i/10.0
    snr_REQ.append(f)

print(snr_REQ)



det_l1 = Detector('L1')
apx = 'IMRPhenomD'
N=2048*16  #N is number of samples, N=length/delta_t
fs=2048 #fs is sampling frequnecy
length= N// fs #duration of segment
delta_f= fs/ N
f_samples = N//2 +1
```

```
f_lower=50
delta_t=1.0/2048

print(length, delta_f, f_samples, delta_t)

from pycbc.psd.analytical import AdVDesignSensitivityP1200087

def get_psd(f_samples, delta_f, low_freq_cutoff):
    psd=AdVDesignSensitivityP1200087(f_samples, delta_f,
low_freq_cutoff)
    return psd

psd=get_psd(f_samples, delta_f, f_lower)

def pure_signal_processing(signal):
  signal.start_time = 0
  signal.append_zeros(4*2048)
  signal = signal.cyclic_time_shift(2)
  signal.start_time = 0
  return signal

def get_peak_time(signal):
  peak = signal.numpy().argmax()
  time = signal.sample_times[peak]
  return time

def get_025s(signal):
    time = get_peak_time(signal)
    cropped = signal.time_slice(time-0.1875,time+0.0625)
    return cropped

from pycbc.noise.gaussian import frequency_noise_from_psd
def get_noise(psd, seed=None):
    noise=frequency_noise_from_psd(psd, seed=seed)
    noise_time = noise.to_timeseries()
    return noise_time

def add_noise_signal(noise, signal):
    noise.start_time = 0
    signal.start_time = 0
    length_signal = len(signal)
    signal_plus_noise=noise

signal_plus_noise[0:length_signal]=np.add(noise[0:length_signal],
signal)
    return signal_plus_noise

def SNR_MF(signal_fin, noise_fin):
  noise_fin.start_time = 0
  signal_fin.start_time = 0
  hps=signal_fin
  conditioned=noise_fin
```

```
  hps.resize(len(conditioned))
  template = hps.cyclic_time_shift(hps.start_time)
  psd_whiten=interpolate(welch(conditioned), 1.0 /
conditioned.duration)
  snr = matched_filter(template, conditioned, psd=psd_whiten,
low_frequency_cutoff=40, sigmasq = 1)
  # pylab.figure(figsize=[10, 4])
  # pylab.plot(snr.sample_times, abs(snr))
  # pylab.ylabel('Signal-to-noise')
  # pylab.xlabel('Time (s)')
  # pylab.show()
  peak = abs(snr).numpy().argmax()
  snrp = snr[peak]
  time = snr.sample_times[peak]
  # print(time)
  # print("We found a signal at {}s with SNR {}".format(time,
  #
round(abs(snrp), 2)))
  return round(abs(snrp),2), time

def scale_SNR(signal, noise, snr_req):
  noise.start_time = 0
  signal.start_time = 0
  snr,_ = SNR_MF(signal, noise)
  signal = (signal/snr) * snr_req
  # final = add_noise_signal(noise, signal)
  return signal

from pycbc.psd import welch, interpolate
def get_whiten(signal_plus_noise):
    signal_freq_series=signal_plus_noise.to_frequencyseries()
    numerator = signal_freq_series
    psd_to_whiten = interpolate(welch(signal_plus_noise), 1.0 /
signal_plus_noise.duration)
    denominator=np.sqrt(psd_to_whiten)
    whiten_freq = (numerator / denominator)
    whiten=whiten_freq.to_timeseries().highpass_fir(30.,
512).lowpass_fir(300.0, 512)
    return whiten

def get_data(signal, corrupted):
  corrupted.start_time = 0
  whiten = get_whiten (corrupted)
  _, time = SNR_MF(signal, whiten)
  # print(time)
  data = whiten.time_slice(time-0.3, time+0.2)
  data.start_time = 0
  data = data/np.std(data)
  return data


def DISTRIBUTIONS(low, high, samples):
```

```
    var_dist = distributions.Uniform(var = (low, high))
    return var_dist.rvs(size = samples)

def SPIN_DISTRIBUTIONS(samples):
    theta_low = 0.
    theta_high = 1.
    phi_low = 0.
    phi_high = 2.
    uniform_solid_angle_distribution =
distributions.UniformSolidAngle(polar_bounds=(theta_low,theta_high),

azimuthal_bounds=(phi_low,phi_high))
    solid_angle_samples =
uniform_solid_angle_distribution.rvs(size=samples)
    spin_mag = np.ndarray(shape=(samples), dtype=float)
    for i in range(0,samples):
        spin_mag[i] = 1.
    spinx, spiny, spinz =
co.spherical_to_cartesian(spin_mag,solid_angle_samples['phi'],solid_
angle_samples['theta'])
    return spinz

def get_params(samples):
    mass1_samples = DISTRIBUTIONS(10, 81, samples)
    mass2_samples = DISTRIBUTIONS(10, 81, samples)
    right_ascension_samples  = DISTRIBUTIONS(0 , 2*math.pi, samples)
    polarization_samples = DISTRIBUTIONS(0 , 2*math.pi, samples)
    declination_samples = DISTRIBUTIONS((-math.pi/2)+0.0001,
(math.pi/2)-0.0001, samples)
    spinz1 = SPIN_DISTRIBUTIONS(samples)
    spinz2 = SPIN_DISTRIBUTIONS(samples)
    snr_req = snr_REQ
    print(snr_req)
    # snr_req = np.int64(snr_req)
    DIST = DISTRIBUTIONS(2500, 3000, samples)
    return mass1_samples, mass2_samples, right_ascension_samples,
polarization_samples, declination_samples, spinz1, spinz2, snr_req ,
DIST


samples = 1700
pure_training_DATA = np.zeros((samples, 512))
noisy_training_DATA =  np.zeros((samples, 512))

def initialise():
  global pure_training_DATA
  global noisy_training_DATA
  pure_training_DATA = np.zeros((samples, 512))
  noisy_training_DATA  =  np.zeros((samples, 512))

def save():
  noisy_dir = '/content/noisy_test_DATA_mix_1' # write the file name
in which you need to put the data
```

```
  np.save(noisy_dir , noisy_training_DATA )
  pure_dir = '/content/pure_test_DATA_mix_1' # write the file name
in which you need to put the data
  np.save(pure_dir , pure_training_DATA)

from astropy.coordinates.distances import Distance
def DATA_GENERATION(samples):

  # initialise()
  mass1_samples, mass2_samples, right_ascension_samples,
polarization_samples, declination_samples, spinz1, spinz2, snr_req,
DIST = get_params(samples)

  for i in range(0,samples):
        seed =  random.randint(1, 256)
        # NOTE: Inclination runs from 0 to pi, with poles at 0 and
pi
        #       coa_phase runs from 0 to 2 pi.
        hp, hc =
get_td_waveform(approximant=apx,mass1=mass1_samples[i][0],mass2=mass
2_samples[i][0],
                                  spin1z=spinz1[i],
spin2z=spinz2[i],
                                  delta_t=delta_t, f_lower=30,
distance = DIST[i][0])

        signal_l1 = det_l1.project_wave(hp, hc,
right_ascension_samples[i][0], declination_samples[i][0],
polarization_samples[i][0])
        signal_l1 = pure_signal_processing(signal_l1)



        # pure_training_DATA[i] = signal_final

        noise=get_noise(psd)
        whiten = get_whiten(noise)
        whiten = whiten/np.std(whiten)


        snr,time = SNR_MF(signal_l1, noise)
        # print(time)
        signal_l1 = (signal_l1/snr) * snr_req[i]


        signal_l1.start_time = 0
        whiten.start_time = 0
        signal_l1 = get_025s(signal_l1)
        whiten_noise = whiten.time_slice(3,3+0.25)

        signal_l1.start_time = 0
        whiten_noise.start_time = 0
```

```
            signal_final = signal_11* (10**22)
            data_final = add_noise_signal(whiten_noise, signal_final)

            # pylab.figure()
            # pylab.plot(data_final.sample_times, data_final)
            # pylab.plot(signal_final.sample_times, signal_final)
            # pylab.legend(('Noisy Signal', 'Pure Signal'))


            # print(len(signal_final))
            # print(len(data_final))
            pure_training_DATA[i] = signal_final
            noisy_training_DATA[i] = data_final

    save()
    print("done")

negative_samples = 2560
negative_training_DATA = np.zeros((negative_samples, 512))

def negative_save():
    negative_dir = '/content/negative_training_DATA' # write the file
name in which you need to put the data
    np.save(negative_dir , negative_training_DATA )

def NOISE_GENERATOR(samples):
    for i in range(samples):
        seed =  random.randint(1, 1000)
        noise=get_noise(psd, seed)
        # pylab.plot(noise.sample_times, noise)
        whiten = get_whiten (noise)
        noise_signal = whiten.time_slice(3,3+0.25)
        noise_signal.start_time = 0
        noise_signal = noise_signal/ np.std(noise_signal)
        # pylab.figure()
        # pylab.plot(noise_signal.sample_times, noise_signal)
        # pylab.show()
        negative_training_DATA[i] = noise_signal
    negative_save()

NOISE_GENERATOR(negative_samples)
```

DAE Model

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g.
pd.read_csv)

# Input data files are available in the read-only "../input/"
directory
# For example, running this (by clicking run or pressing
Shift+Enter) will list all files under the input directory
```

```
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

from google.colab import drive
drive.mount('/content/drive')

import numpy as np
import tensorflow as tf
from keras.layers import Dense, Flatten, MaxPooling1D, UpSampling1D
from keras.models import Sequential
from keras.layers import Conv1D, Dropout
from tensorflow.keras.optimizers import Adam, RMSprop
from keras.losses import BinaryCrossentropy
from keras.metrics import binary_accuracy, AUC
from sklearn.utils import shuffle
import matplotlib.pyplot as plt
from keras.layers import LSTM, RepeatVector, TimeDistributed,
Bidirectional, Dropout, Activation
from keras.constraints import max_norm
import keras

def build_model():

    model = Sequential()
    model.add(Bidirectional(LSTM(256,
        return_sequences = True)))
    model.add(Dropout(0.2))

    model.add(Bidirectional(LSTM(16,
        return_sequences = True)))
    model.add(Dropout(0.2))

    model.add(Bidirectional(LSTM(1,
        return_sequences = True)))
    model.add(Dropout(0.2))
    model.add(Bidirectional(LSTM(16,
        return_sequences = True)))
    model.add(Dropout(0.2))

    model.add(Bidirectional(LSTM(256,
        return_sequences = False)))
    model.add(Dropout(0.2))

    model.add(Dense(512))
    model.add(Activation("linear"))

    model.compile(loss="mse",
optimizer=Adam(learning_rate=0.001,epsilon=1e-07))
```

```
    return model

model = build_model()

postrainsamples = 2000


samples = postrainsamples
print(samples)
pure_training_data =
np.load('/content/drive/MyDrive/FYP/data/pure_training_DATA.npy')
noisy_training_data =
np.load('/content/drive/MyDrive/FYP/data/noisy_training_DATA.npy')

X = noisy_training_data
print(X.shape)
Y = pure_training_data
print(Y.shape)
X = X[:samples]
Y = Y[:samples]
print(X.shape)
print(Y.shape)
X=X.reshape((samples,512,1))
Y=Y.reshape((samples,512,1))
X, Y = shuffle(X, Y, random_state=53)

index = 0
batch_samples = 40
def get_data():
  global X
  global Y
  global index
  global batch_samples
  trainX = np.zeros((batch_samples, 512,1))
  trainY = np.zeros((batch_samples, 512,1))
  index = 0
  while True:
    for i in range(0,batch_samples):
      trainX[i] = X[index+i]
      trainY[i] = Y[index+i]
    index = index+ batch_samples
    if (index>= 1999):
      index = 0
    yield trainX, trainY

from keras.callbacks import ModelCheckpoint
checkpoint_path = "/content/drive/MyDrive/FYP/data/BBH_DAE_1.ckpt"


# Create a callback that saves the model's weights every 5 epochs
cp_callback = ModelCheckpoint(
    filepath=checkpoint_path,
```

```
    verbose=0,
    save_weights_only=True,
    save_freq = 1)

def model_training():
  global samples
  global batch_samples

  model.fit(get_data() , epochs=10, verbose = 1, steps_per_epoch =
(samples//batch_samples), shuffle=
                          True, callbacks=[cp_callback],
batch_size=batch_samples , use_multiprocessing = True)
  model.summary()
  model.save("/content/drive/MyDrive/FYP/data/BBH_DAE_1.h5")

model_training()
```

DAE Model Testing

```
point = 150
p = X[point]
q = Y[point]

import matplotlib.pyplot as plt
plt.plot(range(0, len(p)), p)
# plt.figure()
plt.plot(range(0, len(p)), q)

p = p.reshape((1, 512, 1))
s = model.predict(p)
import matplotlib.pyplot as plt
s = s.reshape((512,1))
plt.plot(s)
plt.plot(q)
!mkdir -p ./FYP/saved_model
model.save('./FYP/saved_model/my_model')
new_model = tf.keras.models.load_model('./FYP/saved_model/my_model')
results = model.evaluate(X, Y, batch_size=40)
print("test loss, test acc:", results)
```

BIBLIOGRAPHY

[1] N. Lopac, F. Hržić, I. P. Vuksanović and J. Lerga, "Detection of Non-Stationary GW Signals in High Noise From Cohen's Class of Time–Frequency Representations Using Deep Learning," in IEEE Access, vol. 10, pp. 2408-2428, 2022, doi: 10.1109/ACCESS.2021.3139850.

[2] H. Shen, D. George, E. A. Huerta and Z. Zhao, "Denoising Gravitational Waves with Enhanced Deep Recurrent Denoising Auto-encoders," ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2019, pp. 3237-3241, doi: 10.1109/ICASSP.2019.8683061.

[3] S. Fan, Y. Wang, Y. Luo, A. Schmitt and S. Yu, "Improving Gravitational Wave Detection with 2D Convolutional Neural Networks," 2020 25th International Conference on Pattern Recognition (ICPR), 2021, pp. 7103-7110, doi: 10.1109/ICPR48806.2021.9412180.

[4] F. U. M. Ullah, A. Ullah, I. U. Haq, S. Rho and S. W. Baik, "Short-Term Prediction of Residential Power Energy Consumption via CNN and Multi-Layer Bi-Directional LSTM Networks," in IEEE Access, vol. 8, pp. 123369-123380, 2020, doi: 10.1109/ACCESS.2019.2963045.

[5] H. Chiang, Y. Hsieh, S. Fu, K. Hung, Y. Tsao and S. Chien, "Noise Reduction in ECG Signals Using Fully Convolutional Denoising Autoencoders," in IEEE Access, vol. 7, pp. 60806-60813, 2019, doi: 10.1109/ACCESS.2019.2912036.

[6] S. Kiranyaz, T. Ince and M. Gabbouj, "Real-Time Patient-Specific ECG Classification by 1-D Convolutional Neural Networks," in IEEE Transactions on Biomedical Engineering, vol. 63, no. 3, pp. 664-675, March 2016, doi: 10.1109/TBME.2015.2468589.

[7] G. Baltus, J. -R. Cudell, J. Janquart, M. Lopez, S. Caudill and A. Reza, "Detecting the early inspiral of a gravitational-wave signal with convolutional neural networks," 2021 International Conference on Content-Based Multimedia Indexing (CBMI), 2021, pp. 1-6, doi: 10.1109/CBMI50038.2021.9461919.

[8] P. Singh, A. Singhal and S. D. Joshi, "Time-Frequency Analysis of Gravitational Waves," 2018 International Conference on Signal Processing and Communications (SPCOM), 2018, pp. 197-201, doi: 10.1109/SPCOM.2018.8724396.

[9] S. Kiranyaz, T. Ince, O. Abdeljaber, O. Avci and M. Gabbouj, "1-D Convolutional Neural Networks for Signal Processing Applications," ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2019, pp. 8360-8364, doi: 10.1109/ICASSP.2019.8682194.

[10] Daniel George, E.A. Huerta, Deep Learning for real-time gravitational wave detection and parameter estimation: Results with Advanced LIGO data, Physics Letters B, Volume 778, 2018, Pages 64-70, ISSN 0370-2693, https://doi.org/10.1016/j.physletb.2017.12.053.

[11] Plamen G. Krastev, Real-time detection of gravitational waves from binary neutron stars using artificial neural networks, Physics Letters B, Volume 803, 2020, 135330, ISSN

0370-2693, https://doi.org/10.1016/j.physletb.2020.135330.

[12] Xia, Heming, Lijing Shao, Jun-jie Zhao and Zhoujian Cao. “Improved deep learning techniques in gravitational-wave data analysis.” *ArXiv* abs/2011.04418 (2020): n. Pag.

[13] Rituerto-González, E., Peláez-Moreno, C. End-to-end recurrent denoising autoencoder embeddings for speaker identification. *Neural Comput & Applic* 33, 14429–14439 (2021). https://doi.org/10.1007/s00521-021-06083-7

[14] S. Singh, A. Singh, A. Prajapati and K. N. Pathak, "Deep learning for estimating parameters of gravitational waves," in Monthly Notices of the Royal Astronomical Society, vol. 508, no. 1, pp. 1358-1370, Aug. 2021, https://doi:10.1093/mnras/stab2417

ACKNOWLEDGEMENTS

I would like to thank all those people whose support and cooperation has been an invaluable asset during the report of this project. I would also like to thank our guide Prof. Dr.M.S.Panse for guiding us throughout the report of this project. It would have been impossible to complete the report of the project without their support, valuable suggestions, criticism, encouragement and guidance. I would like to convey our gratitude also to Prof. Dr. S.J. Bhosale, Head of Department for his motivation and providing various facilities, which helped me greatly in the whole process of this stage of the project. I am also grateful to all other teaching staff members of the Electrical Engineering Department for directly or indirectly helping me for the completion of the Report of the project and the resources provided.