Detection of gravitational waves from LIGO Data

Christopher Lawless

August 2021

Introduction

Gravitational waves are distortions in spacetime, the strongest detectable examples of which come from the collision of black holes.(https://science.mit.edu/big-stories/detecting-gravitational-waves/)

The objective of this project was to use Data Analytics techniques to gather insights into black hole collision detection using LIGO(Laser Interferometer Gravitational-Wave Observatory) Sensor data uploaded to: https://www.kaggle.com/c/g2net-gravitational-wave-detection/data

A Github link to my project can be found here: https://github.com/lawlessc/UCDPA_ChristopherLawless

Blackhole collisions are detected as an increase in frequency in gravitational waves followed by a “ringdown”. The below image is how the waves would look without noise.

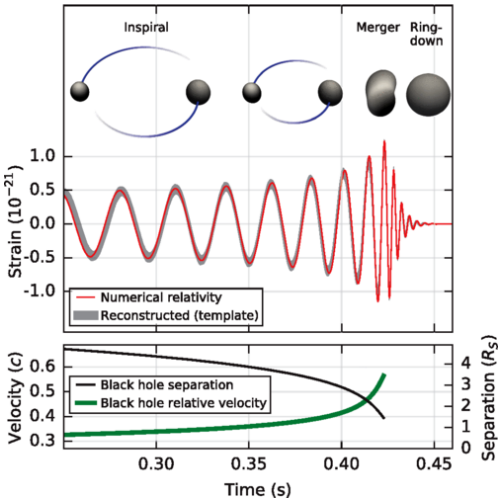

Figure 1: Source https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.116.061102

The Data

This data is synthetic simulations of gravitational waves produced by LIGO that have then had noise applied to them for the purposes of testing different methods of processing and classifying real world gravitational wave data.

The data consists of vibrations detected by three different Gravitational wave sensors in Hanford Washington , Livingston Louisiana, and the Virgo detector in Pisa Italy.

Each sample consists of two seconds of output from all three stored as a 2D NumPy array(.npy) file sampled at 2048hz, or 4096 array items per a sensor, totalling 12288.

From reading the DataFrame of the data on targets i have found there are 560,000 samples
They are evenly split between targets of 1 and 0

For moving the data from files to training i chose to add them to a DataFrame as this made appending the target values and ID's to the data simpler. I also used a DataFrame when i imported target data from **training_lables.csv** for readability and ease of retrieving entries.

```
(560000, 2)
Index(['id', 'target'], dtype='object')
<bound method NDFrame.head of
0       00000e74ad       1
1       00001f4945       0
2       0000661522       0
3       00007a006a       0
4       0000a38978       1
...            ...     ...
559995  ffff9a5645       1
559996  ffffab0c27       0
559997  ffffcf161a       1
559998  ffffd2c403       0
559999  fffff2180b       0
```

Figure 3: An output summary of a DataFrame containing sample ID's and Targets

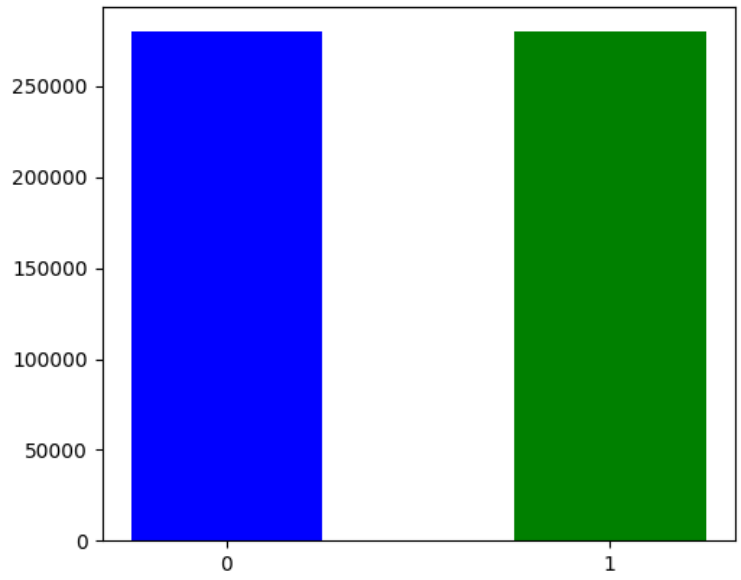

Figure 2: Image shows an even split of positive and negative target examples using count values function

Processing the Data

The data can be downloaded within the project using the function in the kaggle_download.py file. Because this can take a while i have left one folder ‘0/0/0’ in the project files for other users that want to run it immediately.

One thing i did notice from visualisation is that one of three sensor output samples(in green) often appeared to be an inverted version of the signal from the other two detectors. So i inverted this at preprocessing in the hope it will be easier for the neural network to pick up on patterns during training. This is most likely the Italian gravity wave sensor which is located far from the others(And at a different angle).

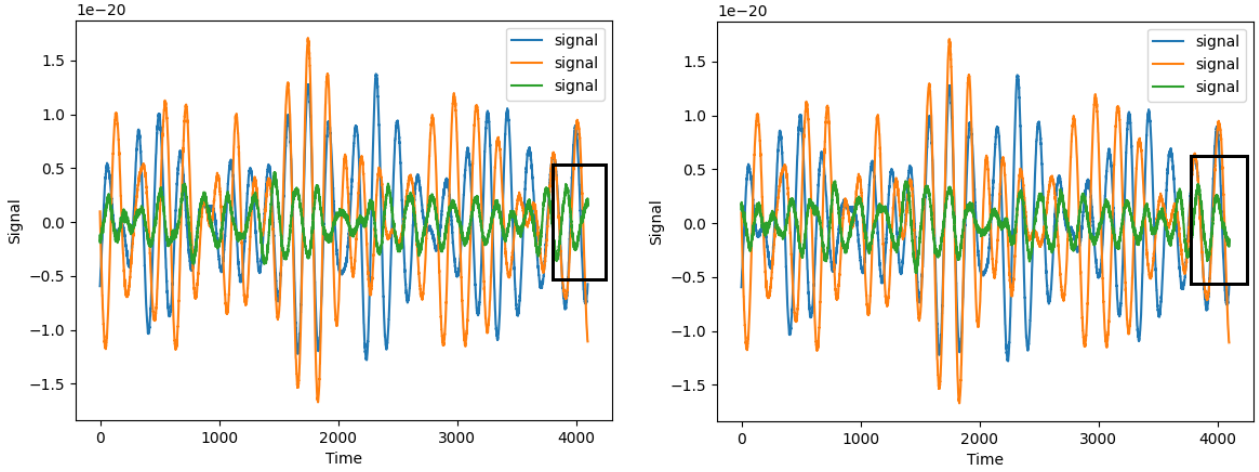

Figure 4: The green signal after being inverted more closely matches the others.(The same issue was mentioned on the competition page)

While working on the project I went through a lot of scaling methods, one of the more interesting things I came across was the Scikit-learn Fast Fourier transform, this is a sklearn feature that allows you to view frequencies ,their amplitudes and even apply a transform to the original signal to remove specific frequencies from the signal. Scipy also provides a function to create a spectrogram of the same signal.

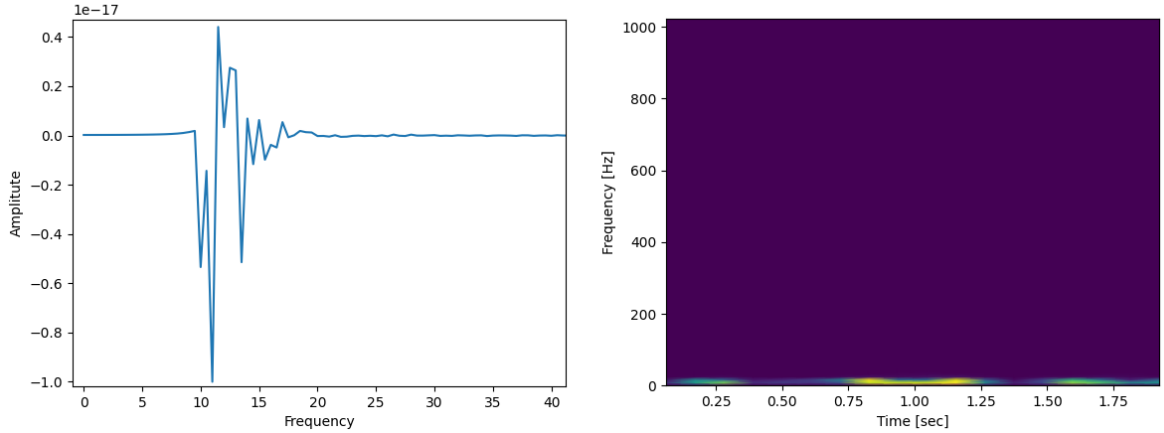

Figure 5: FFT (Left)and spectrogram(Right) of the sensor data. We can see all waves and noise detected are both of a very low amplitude and freqeuncy. The spectrogram shows the same thing but as it changes across time.

The Model

I decided to use a feed-forward neural network ,trained via the Keras API ,because of the large number of parameters which are small floating point numbers . the binary target of the outputs and as they're suited to classification tasks.

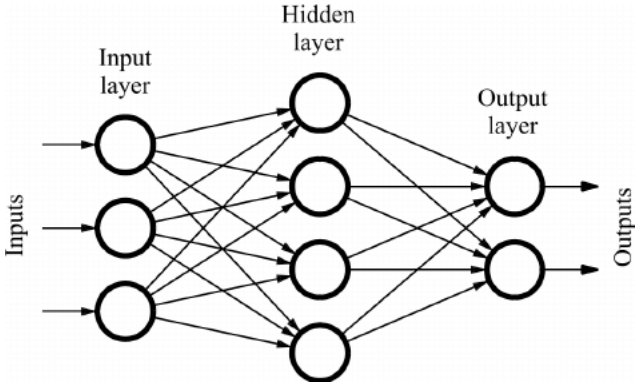

Figure 6: Source: https://www.researchgate.net/figure/Sample-of-a-feed-forward-neural-network_fig1_234055177

I initially started in Keras with three basic Dense layers. The output layer was always either a single Dense layer with a sigmoid activation function or a two neuron layer using softmax. I then moved onto using LeakyRelu layers as i was worried about the possibility of the "dying relu" problem coming up depending on how i scaled my Data. These were much faster at getting through batch or epochs even if the results were the same.

Finally I moved onto trying out convolutional layers in the input and middle layers of my model.

Training the model

Initially i tried to manually train the model, using a class i created called "NeuralNetDefiner" . This however required this required often reloading data every time i wanted to make changes. I then wrote my own hyper-parameter tuning class called

“HyperParameterOpto”, this was more useful as i had set it up to save the best models as it was running, and display training results as a graph.However this was still slow and inflexible. I also found that loss and accuracy often diverged from another by a large amount or both plateaud in similar places.

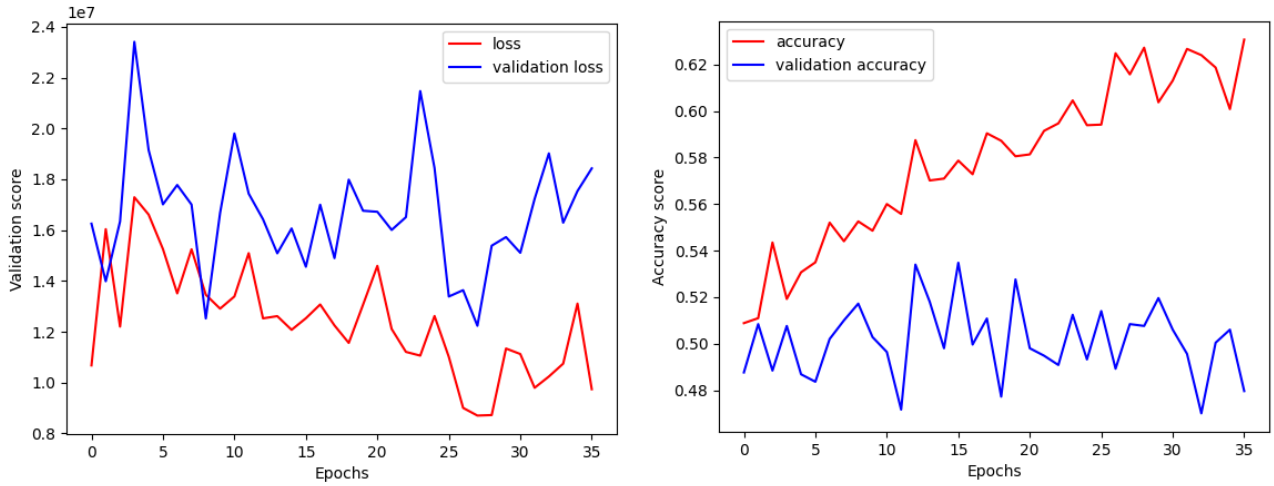

Figure 7: While the training scores(Red) would improve , the validation scores(blue) consistently did not. With accuracy often hovering around 50%

I then implemented GridSearchCV in the “GridSearcher” class within my project. This worked better in the sense it allowed me to try many more options to experiment with multiple layers, optimization algorithms number of epochs and batch sizes. However the over-fitting problem persisted even while using GridSearchCV.

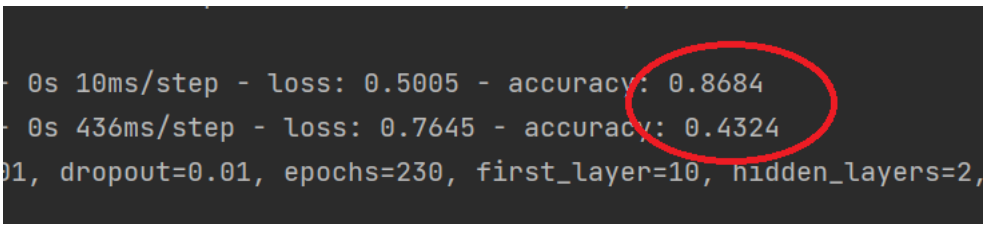

Figure 8: Image demonstrates over-fitting in difference between training and validation accuracy. 1.0 would be perfect accuracy.

Issues with training

When training the models I can over-fit to the point of perfect accuracy with a few hundred to sometimes a few thousand of samples and only several to tens of neurons in the first layer. This does get harder as I increase the number of data points however accuracy remains at ~50%. Some perform slightly above and below 50% , this seems to be because data reserved for testing doesn't have a perfectly even distribution of positive and negative examples.

Possible causes

I considered multiple causes for this.

1. Buggy code when importing data causing targets to be matched to the wrong samples
2. Driver/Cuda setup error or hardware error in my GPU/CPU
3. The data is too complex and noisy
4. Improper scaling

1. I followed up on this issue by checking the contents and format of my data at every stage between importing and beginning training, and did find issues with incorrect targets

So i fixed these bugs , but found the over-fitting issue remained. I also made sure to load the data as 32 bit floats and convert the targets to the same to avoid possible issues Keras might have

2. I believe the fact the network can over-fit and the lack of error messages from Cuda/Keras means that my hardware and drivers are ok.

3. For this project I understood ,too late, just how noisy the signals are. I believe this is the most likely issue. The models i have tried easily learn the noise in small sets of training data

Possible Solutions:

1. The data needs to be preprocessed more to remove noise before models train or make predictions. This could maybe be done with a Fast Fourier transform , I did experiment with this via the Sklearn library and saw what may be the signal of one in the data labelled as Target 1. That is often absent in the Target 0 samples.

   These same scaling or transformation methods would also need to be applied to any new data from which predictions are made, having the disadvantage of making any model more complex.

2. Training the networks for longer on larger datasets. The total number of samples in the training set is 560,000 ,which is likely large enough. However I was never able to go beyond using 60,000 samples from the training sets in my systems memory. This could be fixed by breaking training into multiple sessions and reloading the data between those sessions.

3. Using Ensembles. I initially considered trying to use an Ensemble of networks, but these would only be useful if the models had an accuracy greater than 50% .
4. I could also training three different networks one for each sensor. And if just one of them works, train that model also with data from the other sensors(transfer learning)

5. I tried adding dropout layers to prevent over fitting. However this may have been futile as the data itself is already very noisy

6. Analysis of the data in more ways such as how various scaling methods affect it or for example PCA (Principal Component Analysis) to reduce the dimensionality of the problem and thus the required number of inputs.

7. The user of other classification methods like K-means clustering

8. Better use of scaling, as i can’t be certain some methods of scaling aren’t removing information from the signal that is necessary for classification

9. Adding layers to the network that allow me to view model output during training and validation testing, not a direct solution but could generate insights

Imports used in this project:

Sklearn (fft,scaling)

Scipy.signal (Spectrograms)

Keras (Neural Networks)

TensorFlow (Backend for building and training Neural Networks)

Matplotlib (visualizers)

Matplotlib.pyplot (visualizers)

Kaggle (Retrieving Competition Data)

This project is also using Python Version 3.7 and CUDA

The G2net dataset https://www.kaggle.com/c/g2net-gravitational-wave-detection/data

Pycharm was used as the IDE