G2Net Detecting Continuous Gravitational Waves

Enric Reverter
Pim Schoolkate

January 2023

1 Introduction

The problem to tackle is the detection of continuous gravitational-wave signals (CW), which is proposed as an ongoing competition in *Kaggle* [1]. There are four classes of waves, yet at present only signals from merging black holes and neutron stars have been detected. The gravitation waves from these sources were first detected and discovered in 2015 for which the Nobel Prize in physics was awarded. Continuing on this research, the scientists from G2net aim to detect the above-mentioned continuous gravitational-wave signals by means of Machine Learning. As of now, such waves have been detected using an algorithm named SOAP [2], which is based on the Viterbi algorithm. Also, machine learning models have been applied on top of it. Specifically, a Convolutional Neural Network (CNN) [3]. The state of this search was reviewed in 2021 [6].

2 Motivation

The motivation for doing this particular project is the possibility to add to ongoing and cutting edge research, while also challenging the skills of the students. It is not expected that a significant impact to the problem can be made, however, that should be a reason not to try.

3 Data

The data provided is from two gravitational-wave interferometers (LIGO Hanford & LIGO Livingston). Such tools detect and merge different sources of light to create inference patterns that can later be analyzed. The data is stored as time-frequency sets, where the aim is to detect whether a CW is present or not. That is, a binary outcome.

Each sample is comprised of a set of Short-time Fourier Transforms (STFTs) and corresponding GPS time stamps for each interferometer. Frequencies are also stored, but they are shared by both detectors. Also, the STFTs are not always contiguous in time, since the interferometers are not continuously online.

It is worth to mention that the authors only provide a small training set, because it is expected to generate more data by means of *PyFstat*, a library specifically build to generate data.

Since data is stored in a set of STFTs, both temporal and frequency domains can be considered for the analysis. That is, both time-series and computer vision approaches can be taken. However, only the latter has been tackled in this project.

Three sources of data are used for finding the best models. First, the training set provided by the competition, which only contains generated samples with Gaussian noise, not the real observations

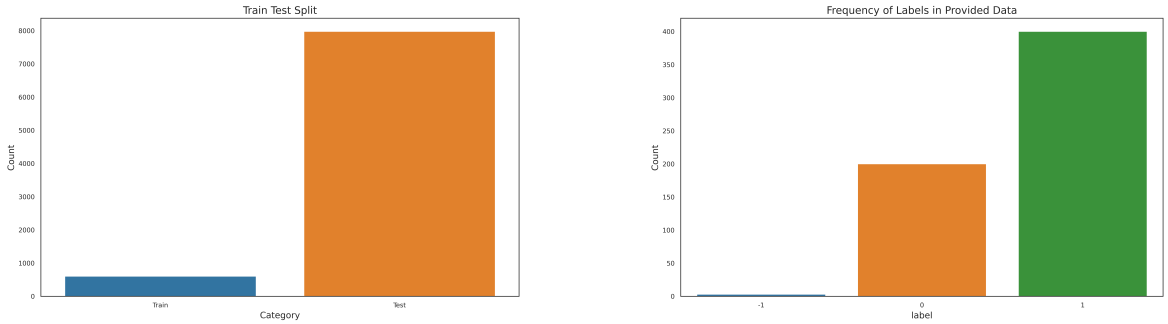

Figure 3.1: At LHS, the training and test samples available. At RHS, the label frequencies of the training samples.

from the interferometers. Second, data is generated by means of *PyFstat*, which again only contains samples with Gaussian noise. Third, samples with simulated real noise are generated by copying temporal dynamics that can only be observed in some instances from the test set (i.e., real noise captured by the detectors).

3.1 Provided Data

As above-mentioned, 200 GB of data are provided for the competition. However, almost all is meant for the test set, as depicted in Figure 5.3. The train set consists of around 600 samples, the test set around 8000. Each file is a Hierarchical Data Format version 5 (*hdf5*), which allows to store compressed data in different groups. In particular, the provided files consist of a dataset containing the frequencies registered by each detector, and one group for each of them with additional datasets about the GPS timestamps and the STFTs amplitudes. There is an additional dataset containing the labels of the training samples, which distribution can be depicted in Figure 5.3. Note how the outcome is binary, but there are three labels: "0", "1", and "-1". The "-1" is referred to those observations for which the competition holders do not know the real label. They are removed since nothing can be inferred from them. It is also worth to mention that all training samples have been generated with *PyFstat*, so the noise is Gaussian in all and there is no missing data.

For the work done in this analysis, only the datasets with STFTs are needed, for which data will be further compressed as discussed in Section 3.4. The data can be visualized as a spectrogram in Figure 3.3. One is depicted for each of the possible labels, 0 (noise) or 1 (signal). More on the details of such spectrograms is discussed in the next section [3.2].

3.2 Generated Gaussian Data

There is a need to generate more data due to the present imbalance observed in the provided data [3.1]. For that, the *PyFstat* library is used. It allows to simulate CWs on top of generated noise by providing a set of parameters. Two configurations need to be provided to the `writer` (the method that writes data into memory), one for the noise and one for the signal. The main parameters to consider for the STFTs are: the Fourier transform time duration, fixed at 1800 seconds; the window type, set as Tukey, also known as the cosine-tapered window; and its associated parameter $\alpha \in [0, 1]$, fixed at 0.01.

When only noise is generated, the parameters to consider are: the central frequency of the band, the band-width around such frequency, the single-sided amplitude spectral density of the noise, and the observation duration. When the signal is also simulated, there are two important considerations, what is the shape and how strong is it with relation to the noise. The shape parameters [4] can be

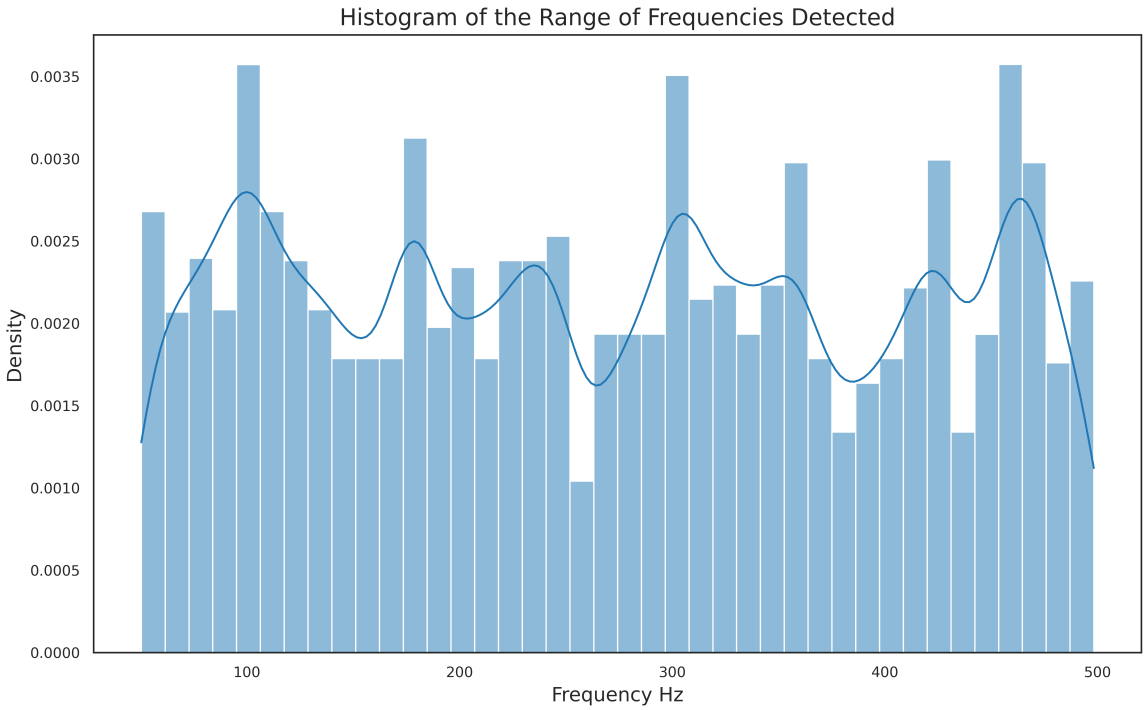

Figure 3.2: Frequency distribution of the data. Note how it is bounded by 50 and 500.

drawn from a prior distribution available in *PyFstat*, so the focus is on its strength. This is quantified by the sensitivity depth, which is defined as the ratio between the noise and CW amplitude:

$$D = \sqrt{\frac{S_n}{h_0}}$$

According to the researchers, visual features tend to disappear around a depth of 20 $Hz^{-\frac{1}{2}}$. For that, the sensitivity depth is drawn from a uniform distribution ranged between 5 and 20 $Hz^{-\frac{1}{2}}$. Figure 3.4 depicts a sample of noise and two of signals, where it can be clearly seen how the sensitivity depth can effect the generated signals. There are lots of stochastic processes involved, for which a random generator in `numpy` with `seed=42` in applied to ensure reproducibility. The number of generated samples is around 5000. Each sample contains the same information that can be found in the training set, and they are also stored as *hdf5* files for the time being.

3.3 Simulated Real Data

The community detected an issue with the training data, which is the fact it only contains Gaussian noise, while the test data has some samples of real noise captured by the detectors. For that, the observations containing real noise have been simulated by inference from those. Note how the test samples are not used for anything besides simulating background noise. That is, the model is not found by using test samples as the signals are embedded a posteriori of getting such noise. The details are explained by Vladimir Slaykovsky [5]. In total, around 1500 samples are generated in this context. An example can be depicted in Figure 3.5, where it can be clearly seen the noise is much more different than the one seen in the previous sources. It is worth to mention this data are directly loaded as *png* files. Moreover, they are black and white (i.e., only one channel). In other words, information about

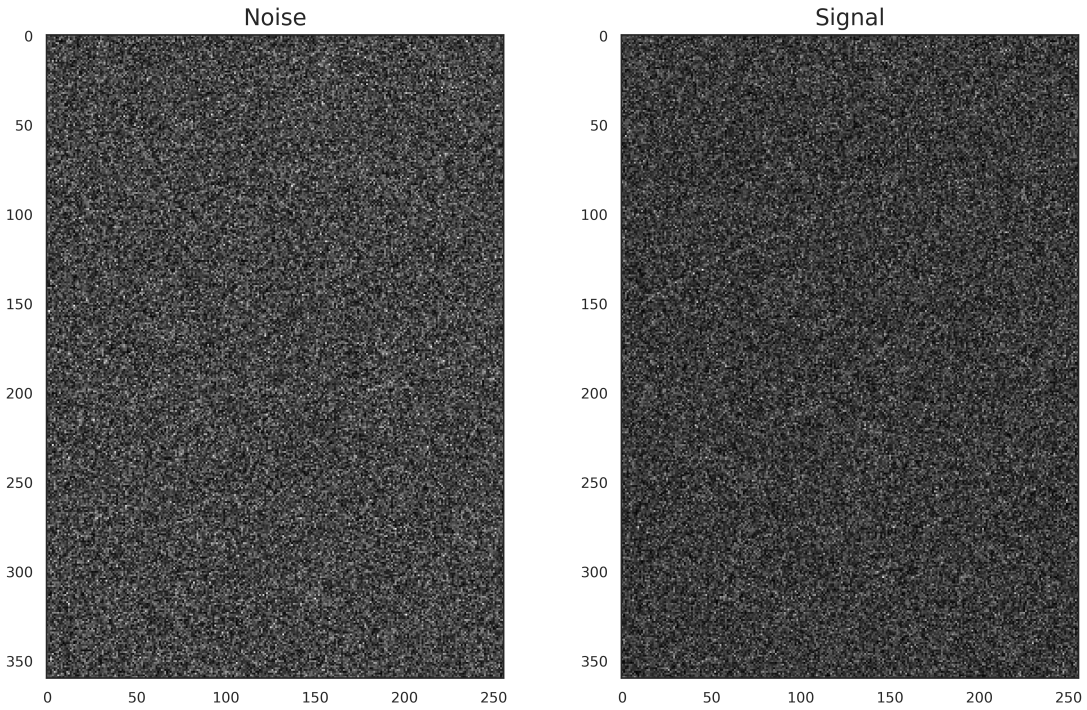

Figure 3.3: Noise and signal spectrograms drawn from the provided training set.

the timestamps, frequencies, and amplitude values is lost. Everything is already normalized between 0 and 255 in (360, 256) arrays.

The specific way in which this samples are obtained is as follows. First, a non-stationary noise sample is loaded. Second, a set of random pure signals (i.e., devoid of background noise) are generated. Third, the signal is added on top of the noise, and multiplied by a factor that allows to control its strength in comparison to the noise in a similar way to the sensitivity depth measure.

3.4 Data Transformations

In order to fit data in memory and train the algorithms data needs to be compressed. STFTs from the training set have a shape of (360, #). That is, the height of the spectrogram is always 360 while the width depends on its duration. Generated data does not have a fixed shape, but it is generally much larger than that (e.g., (3150, 3600)). In order to have standardized data, both sources need to be compressed as the simulated non-Gaussian noise data (i.e., convert them to *png* files of shape (360, 256)).

Starting from STFTs of different shapes, which values are of the 10^{-22}-order, three things need to be accounted for: height needs to be trimmed, width needs to be trimmed, and normalization is required. In the case of training samples, the height is already 360, so compressing it is not necessary. For generated samples, however, not only needs to be trimmed, but also another detail needs to be considered. All samples with signals are generated in such a way that the signal is centered. For that, the horizontal cut is done with a small deviation from the center so the signal can also be detected at the bottom and top of the spectrogram (if possible). Then, width is reshaped without issues. Finally, the values are normalized to those of a single channel *png* file.

Once data has been compressed, normalized, and stored, the validation strategy is considered.

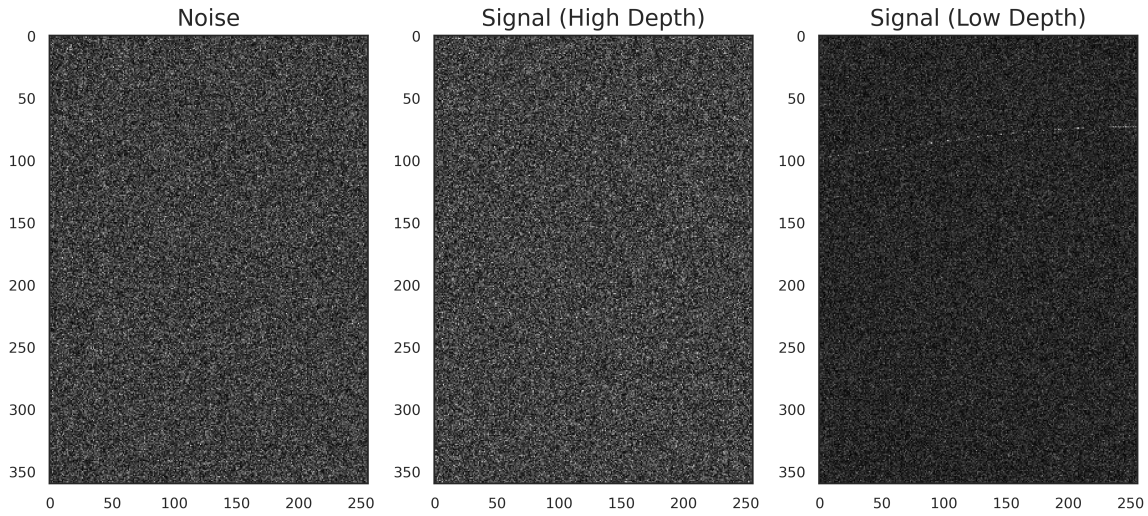

Figure 3.4: Noise and signal spectrograms drawn from the generated set with stationary noise.

First, data is split into stratified train (90%) and test (10%) sets by means of the `train_test_split` method in *scikit-learn*. Then, two different approaches are needed for validation and obtaining the final models. As discussed in Section 4, an exhaustive search over a grid of parameters is computed to find the most suiting parameters of different estimators. The issue arises when the RAM is not enough (30GB) to carry such a search. So the data used for the single folded validation in this strategy is further reduced to a 33% of the training data, roughly 2129 observations to train and 4257 to validate (i.e., the validation set is larger than the training during the exhaustive search). Once the suitable parameters are know, the algorithms are refit using 67% of the training data. That is, now 4257 observations are used to train and 2129 to validate. For the final results, all the training data is used to predict on the test set.

The validation and performance are assessed by means of the F1 score,

$$F1 = 2\frac{(P\dot{R})}{P+R}$$

where P stands for precision,

$$P = \frac{TP}{TP+FP}$$

R stands for recall,

$$R = \frac{TP}{TP+FN}$$

and TP, FP, and FN stand for true positives, false positives, and false negatives, respectively.

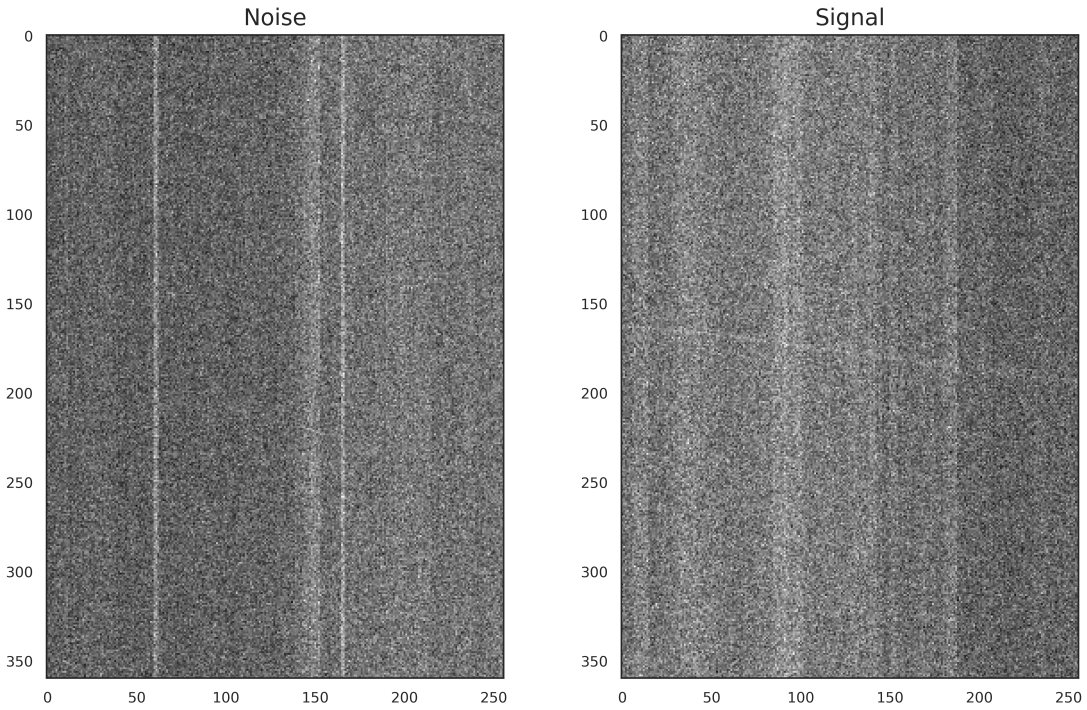

Figure 3.5: Noise and signal spectrograms drawn from the generated set with non-stationary noise.

More precisely, a weighted version of the F1 score is considered, to account for the slight imbalance that might be present between positive (1) and negative (0) labels.

4 Modelling

The considered methods are briefly explained in this section. Two algorithms are applied to find the best model. However, a Naive Bayes classifier is trained first as a baseline for comparison. Then, both Support Vector Machines and Neural Networks are fitted. The results of the methods described here are depicted in Section 5.

4.1 Naive Bayes Classifier

A Gaussian Naive Bayes algorithm is used. It works by assuming conditional independence between the features. Then, the likelihood of the features with which to update the probabilities is assumed to be Gaussian:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

where the values of σ_y and μ_y are estimated using maximum likelihood.

It is implemented in Python with the `GaussianNB` class from *scikit-learn*.

4.2 Support Vector Machine

Support Vector Machine algorithm in its dual formulation is considered. To do so, the `SVC` class from *scikit-learn* is used. If the primal was to be used instead, the `LinearSVC` class would be the choice. Three kernels are tested: linear, polynomial, and radial basis function (RBF).

The **linear kernel** is defined as:

$$K(x, y) = x^T y$$

where x and y are vector in the input space.

The **polynomial kernel** is defined as:

$$K(x, y) = (x^T y + c)^d$$

where x and y are vector in the input space. c is a coefficient that effects the influence of the different order terms. d is simply the degree of the polynomial.

The **radial basis function kernel** is defined as:

$$K(x, y) = \exp\left(-\frac{\|x - y\|^2}{2\sigma^2}\right)$$

where σ is a free parameter.

In order to find the best SVM different configurations of the parameters need to be taken into account. To do so, a `GridSearchCV` is implemented with the following grid of parameters:

```
svm_linear = {'C': [0.01, 1, 100],
              'kernel': ['linear'],
             'class_weight': ['balanced']}
svm_others = {'C': [0.01, 1, 100],
              'gamma': [1, 0.01, 'auto'],
              'class_weight': ['balanced'],
              'kernel': ['poly', 'rbf']}
parameters = [svm_linear, svm_others]
```

Listing 1: Parameters' grid of the different kernels.

Due to the size and complexity of the data, validation is only applied one time for each configuration. Once the seemingly best parameters for each kernel are found, they are refit with more data to check its performance on the test set.

4.2.1 Mixture of Kernels

Finally, a mixture of the above-mentioned kernels is considered. The idea is to see if the addition of them can somehow beat their performance separately. It is worth to mention that a kernel could be implemented from scratch if it was not for the hardware limitations. Using plain Python is not efficient enough for the magnitude of this data. For instance, programming a linear kernel with the `numpy` dot product is unfeasible due to too much RAM being required. For that, a weighted sum of kernels is built as follows:

$$K(x, y) = \alpha * K_{linear} + \beta * K_{poly} + (1 - \alpha - \beta) * K_{rbf}$$

where α and β are the weights associated to the linear and polynomial kernels, respectively. The weight of the radial basis function kernel is simply the complementary value to reach 1 in the sum of weights. Note how this is a valid kernel due to their properties.

The way to implement it is by using the set of existing pairwise kernels in *scikit-learn*. Those are built on top of C and are extremely efficient in comparison to its implementation on vanilla Python, which as mentioned is unfeasible due to RAM limitations. For that, `linear_kernel`, `poly_kernel`, and `rbf_kernel` are used. Then, in order to find the best configuration of parameters, a class is built on top of `SVC`, which allows the implementation and fine-tuning of this mixture. The grid of parameters considered for α and β is the following:

```
parameters = [{'alpha': [0], 'beta': [0.5]},
              {'alpha': [0.5], 'beta': [0]},
              {'alpha': [0.33], 'beta': [0.33]},
              {'alpha': [0.2], 'beta': [0.6]},
              {'alpha': [0.6], 'beta': [0.2]},
              {'alpha': [0.2], 'beta': [0.2]}]
```

Listing 2: Parameters' grid of the mixture of kernels.

The results are discussed in Section 5.

4.3 Convolutional Neural Network

4.3.1 Denoising stacked autoencoder

A stacked denoising autoencoder was considered for this prediction task. Since the raw signal is available from the generation of the data, it is possible to train a model which removes the noise from the images and leaves only the raw signal. The idea is thus to use the output of said denoising model as the input for a classification model. This could be a convolutional network, or any other binary prediction model.

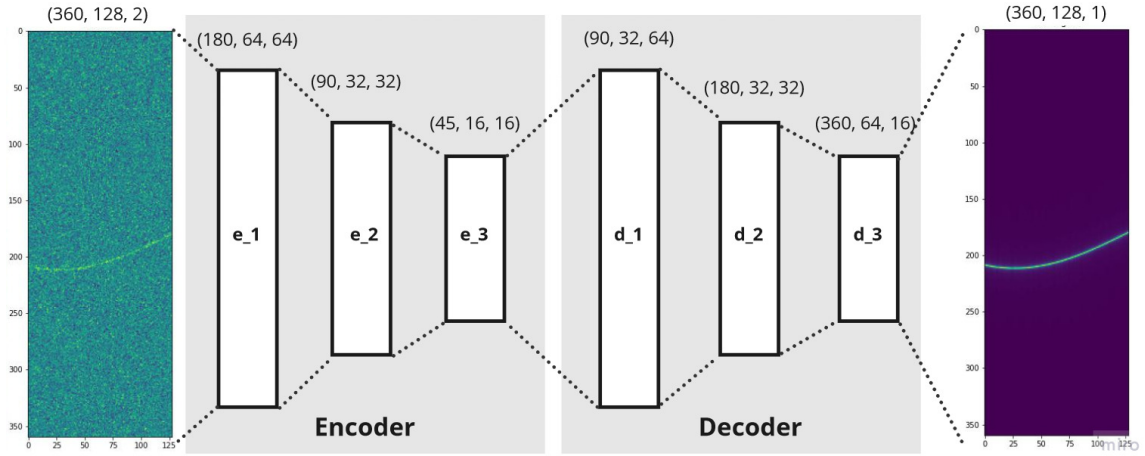

Figure 4.1: Autoencoder architecture.

The idea of the stacked denoising autoencoder is fairly simple. The architecture of one autoencoder is build up out of two components, the encoder and decoder. The encoder attempts to represent the input data in a lower dimensionality, keeping the most important features, in essence, it is a feature extractor The decoder attempts to reconstruct the data, using only the features provided by the encoder. Thus the encoder is trained to provide all the necessary data to reconstruct the original data.

Instead of actually reconstructing the data, in this case, it is only desirable to reconstruct part of the data, the signal. Since the problem allows for generation of the noise and the signals separately,

block	Layers
e_1	Conv2D(filters=64, kernel size=(3,3), activation = "relu", regularization=L2) MaxPooling2D(downsizing=(2, 2)) BatchNormalization
e_2	Conv2D(filters=32, kernel size=(3,3), activation = "relu", regularization=L2) MaxPooling2D(downsizing=(2, 2)) BatchNormalization
e_3	Conv2D(filters=16, kernel size=(3,3), activation = "relu", regularization=L2) MaxPooling2D(downsizing=(2,2))
d_1	Conv2D(filters=64, kernel size=(3,3), activation = "relu", regularization=L2) UpSampling2D(upsizing=(2, 2))
d_2	Conv2D(filters=32, kernel size=(3,3), activation = "relu", regularization=L2) UpSampling2D(upsizing=(2, 2))
d_3	Conv2D(filters=16, kernel size=(3,3), activation = "relu", regularization=L2) UpSampling2D(upsizing=(2, 2))
Out	Conv2D(filters=1, kernel size=(3,3), activation = "sigmoid", regularization=L2)

the input of the autoencoder can be the combination of the signal and noise, like it would be observed by a detector, and the target output can be the isolated signal.

Figure 4.1 shows an overview of the autoencoder architecture used in this problem. `e_1`, `e_2`, and `e_3` Denote the different blocks in the encoder, whereas `d_1`, `d_2`, and `d_3` denote the decoder blocks. Table **??** gives an overview of the blocks and what they include. Each convolutional layer is regularized with L2 regularization to reduce overfitting on the training samples. The `MaxPooling2D` and `UpSampling2D` layers are simply layers to decrease and increase the aspect ratio of the images. The `BatchNormalization` layer is used to normalize over the batches and make them equally important during training.

Each autoencoder is trained separately, using as an input the image and the output of the previous autoencoder. The first autoencoder only has the image as an input. The target variable for each autoencoder is the raw signal. By chaining their inputs and outputs, the autoencoders are essentially stacked.

Choosing a loss function for training the autoencoder is, in this case not straight forward. The images are of size `(360, 128, 1)`, making up for 46,080 pixels to be predicted and as the actual signal only accounts for a small proportion of these the target variable is extremely sparse. For this reason, a mean squared error loss is not viable, as the amount of correct predictions where the pixels should be 0 overshadow the incorrect predictions where the pixel should be 1 in the calculation of the loss. Thus the model would converge to predicting everything as a 0.

For this reason, the cosine embedding loss was used, which computes the cosine distance between two vectors:

$$\alpha_{AB} = \frac{a * b}{|a| * |b|} = \sum \frac{a_i b_i}{\sqrt{\sum a_i^2}\sqrt{\sum b_i^2}} \tag{1}$$

With this, the accurate prediction of the signal has a larger impact on the loss since the distance between a vector of zeros and a vector with sparse ones has just as large of a loss as random predictions, whereas the loss of a correctly predicted signal is close to zero.

The first autoencoder was trained with 2000 generated images either with Gaussian noise or

block	Layers
c	Conv2D(filters=16, kernel size=(3,3), activation = "relu", regularization=L2)
m	MaxPooling2D(downsizing=(3,3)) Dropout(0.5)
out	Dense(1, activation = "sigmoid")

with non-stationary noise as described above, over 20 epochs. For the second autoencoder, the first autoencoder was used as a base and was set to not be trainable. The second autoencoder was trained on a different set of 2000 generated images with either Gaussian or non-stationary noise, in an effort to reduce overfitting on the previous set of images, as well for 20 epochs.

As a test set, the save test data from the SVM part was used.

4.3.2 Binary convolutional head

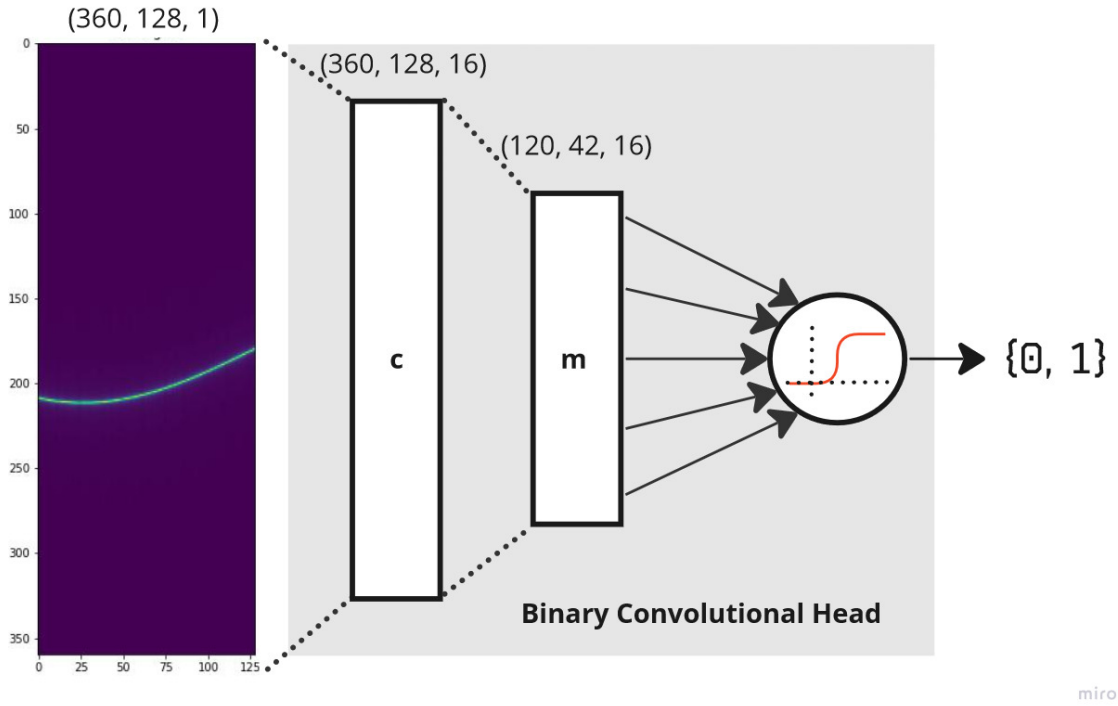

Figure 4.2: Binary Convolutional Head architecture.

The prediction part of the full model takes as an input the denoised signal from the autoencoder. The only detection necessary here is the presence of the signal or not, for which a single convolutional layer, followed by a dense layer with a single neuron was deemed sufficient. The architecture of this model can be found in figure 4.2. Table **??** elucidates the architecture.

Because of results obtained from training a single autoencoder and a stacked autoencoder it was decided to use only the single autoencoder as the base of the model.

The head of this model was trained using the same train and test split as on the SVMs, using a Mean Squared Error, over 20 epochs. The only difference was the use of a validation set during training which amounted for 10% of the training data.

5 Results

5.1 Machine Learning Approach

First, all algorithms in this section have been trained with a flattened version of the data. That is, the arrays of size (360, 256) are reshaped to (92160,). This results in extremely sparse arrays, for which SVMs are not well suited by default. The exhaustive search of parameters for the SVMs with different kernels can be depicted in Table 1. It can be easily seen how the linear kernel outperforms the polynomial and radial basis function kernels. Different values of both C and *gamma* do not change the predictions made by the SVM neither with the linear kernel nor the polynomial one. Then, RBF either predicts everything as noise or a signal, thus the difference in the F1 score. Nonetheless, the models chosen to be refit with more data are the ones depicted in Table 2. One model is trained for each detector, which seems to effect only the results of the linear kernel, where L1 predictions are better. Note how the value of C for the RBF kernel is not specified. This is because more values of it have been tested for the kernel, but none avoids it to predict everything of the same class. Also, the RBF kernel takes longer to compute than the other two, for which it will not be refitted to predict over the test data.

Another exhaustive search is computed for the mixture of kernels model, which results can be depicted in Table 3. The results seem to be consistent for different parameters, although using $C = 0.1$ slightly outperforms the rest of the models for same αs and βs. Giving the same weight to each kernel produces the best score, which is surprising given the bad performance of RBF by itself. For that, the SVM using mixture of kernels with $\alpha = 0.33$, $\beta = 0.33$, and $C = 0.1$ is going to be retrained with more data to be compared using test set predictions.

The results on the test set are depicted in Table 4. For some reason, the algorithms trained for the L1 detector have a better performance that those of the H1. However, when ensembling them together, the results are the same as for the NB classifier, which is not good. The polynomial kernel seems to be the one that outperforms the rest, but they are all equally bad.

5.2 Neural Network Approach

5.2.1 Autoencoder

Both the single and the stacked autoencoders seemed to be promising in that, from a visual analysis, they are able to isolate the signals found in the validation set. Figures 5.4, and 5.5 show two cases of signals correctly isolated from the noise. Only for cases where the signal is extremely weak, the autoencoders were not able to isolate the signal as can be seen in figure 5.6. As can be noted, the predicted signals for the single autoencoder (A1) actually contain less noise than the stacked autoencoder (A2).

However, in the case of the test set, which does not come from the same generated training data, results are worse, as can be seen in figures 5.7 and 5.8.

5.2.2 Binary Convolutional Head

The results for the complete model are not promising. Table 4 shows that the model gets outperformed by all the other approaches.

5.3 Competition Winner Approach

As mentioned in the Introduction 1, this problem has been posed as a *Kaggle* competition, which ended the 2nd of January. The best approaches either involve pre-trained image models or no machine

kernel	C	gamma	Weighted F1 Score
linear	0.01		0.511
linear	100.00		0.511
linear	1.00		0.511
poly	100.00	1	0.485
poly	0.01	1	0.485
poly	100.00	auto	0.485
poly	0.01	0.01	0.485
poly	0.01	auto	0.485
poly	100.00	0.01	0.485
poly	1.00	1	0.485
poly	1.00	0.01	0.485
poly	1.00	auto	0.485
rbf	100.00	0.01	0.346
rbf	100.00	1	0.346
rbf	1.00	1	0.346
rbf	1.00	auto	0.346
rbf	1.00	0.01	0.346
rbf	100.00	auto	0.346
rbf	0.01	auto	0.321
rbf	0.01	0.01	0.321
rbf	0.01	1	0.321

Table 1: Exhaustive search of parameters for SVM. One fold has been used for training and another one for validation.

kernel	C	gamma	Detector	Weighted F1 Score
linear	0.01		H1	0.499
			L1	0.52
poly	0.01	'auto'	H1	0.493
			L1	0.493
rbf	c	'auto'	H1	0.351
			L1	0.351

Table 2: Performance of the best models found by the exhaustive search on a larger training and validation sets. The data of the two detectors is handled separately.

		Weighted F1 Score	
alpha	beta	C=0.0001	C=0.1
0.33	0.33	0.487968	0.508793
0.20	0.20	0.487968	0.508793
0.60	0.20	0.492664	0.507105
0.50	0.00	0.501232	0.505746
0.20	0.60	0.484898	0.504185
0.00	0.50	0.489921	0.501838
0.50	0.50	0.477934	0.503942

Table 3: Exhaustive search of parameters for the SVM with a mixture of kernels. One fold has been used for training and another one for validation.

Model	H1 wF1	L1 wF1	Ensembled wF1
NB	0.470	0.471	0.471
SVM (linear)	0.487	0.527	0.471
SVM (poly)	0.491	0.529	0.472
SVM (mk)	0.483	0.502	0.451
CNN			0.448

Table 4: Test set results. Note how the CNN has been directly trained with the observations from both detectors. *wF1* stands for weighted F1 Score.

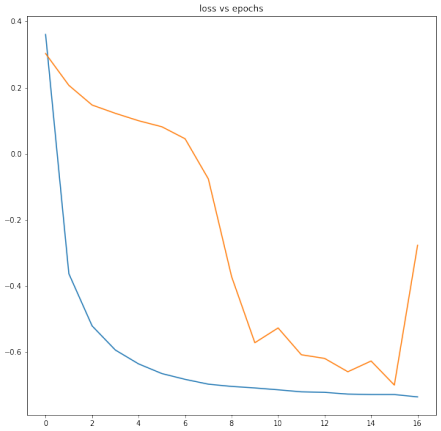

Figure 5.1: Loss vs Epochs for Autoencoder 1

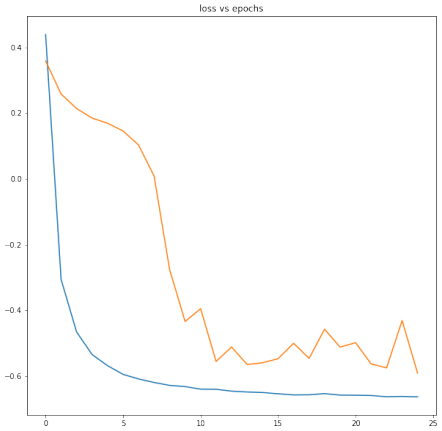

Figure 5.2: Loss vs Epochs for Autoencoder 2

Figure 5.3: Training metrics over epochs. The blue line is the training data, and the orange line the validation data

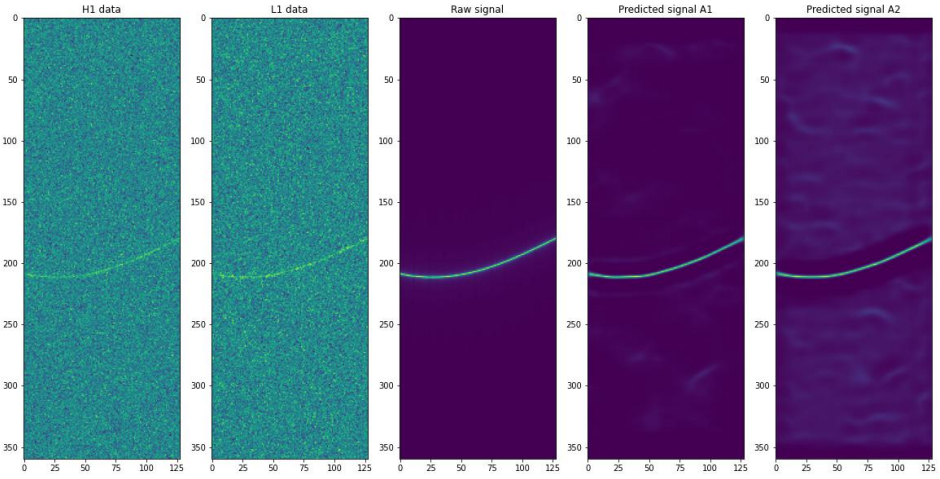

Figure 5.4: Autoencoder validation prediction example with Gaussian noise.

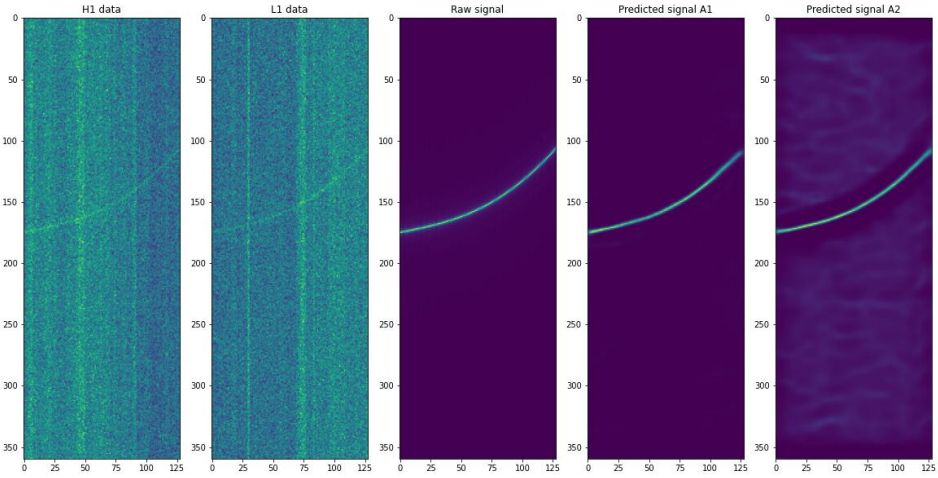

Figure 5.5: Autoencoder validation prediction example with non stationary noise.

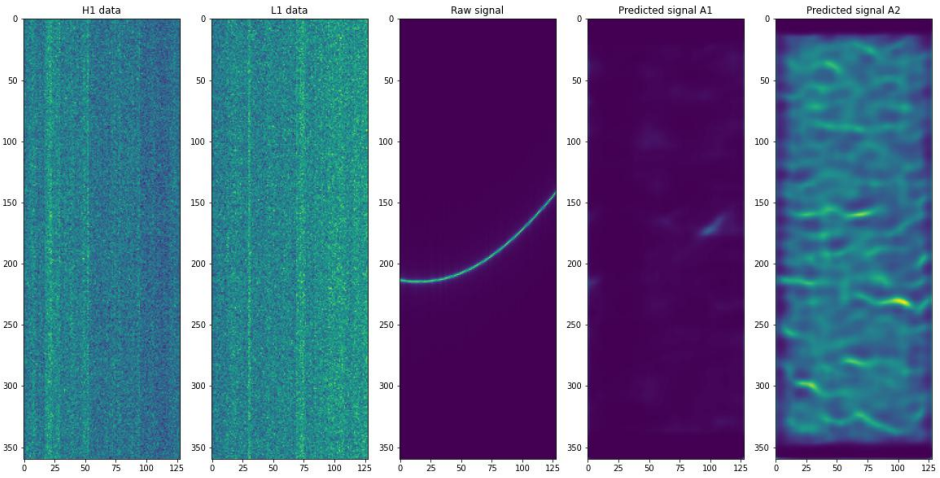

Figure 5.6: Autoencoder validation prediction with weak signal.

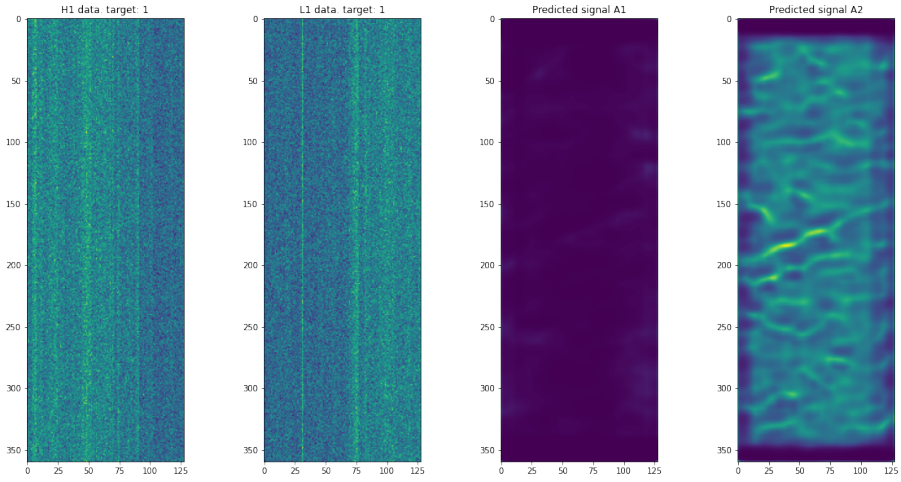

Figure 5.7: Autoencoder test prediction.

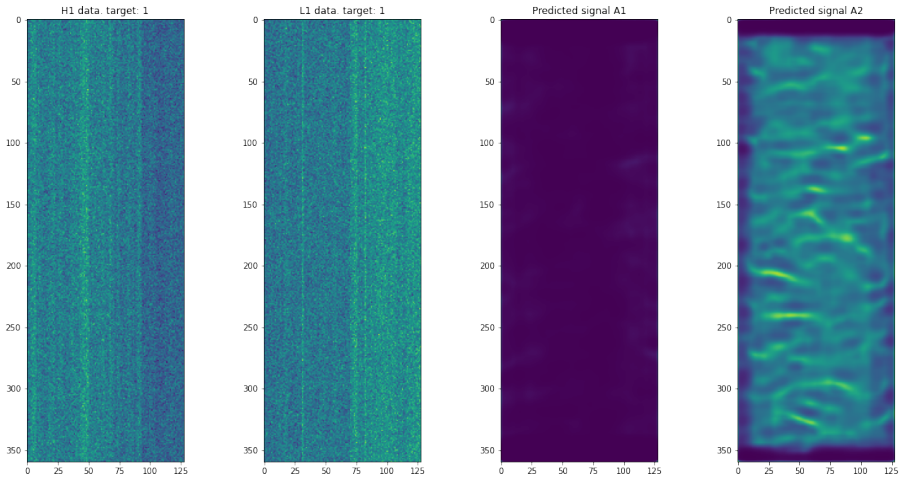

Figure 5.8: Autoencoder test prediction.

learning whatsoever. CNNs obtained decent results compared to other methods as well. In fact, the most elegant solution, which is the winner, does not involve machine learning algorithms. It consists on taking the power summation from the frequencies and amplitudes in the simulations, and then applying a sinc kernel on top. Such kernel is not implemented in *scikit-learn* and the `numpy.sinc` function overflows the RAM, otherwise it could have been a nice implementation to the project.

6 Conclusions and Limitations

The detection of continuous gravitational-waves has been tackled with different algorithms. First, a Naive Bayes (NB) classifier has been used as the baseline to compare. Second, Support Vector Machines (SVM) with different kernels have been trained. A mixture of kernels has been implemented satisfactorily. Finally, a Convolutional Neural Network (CNN) with autoencoders has been fitted. There is a limitation in the number of training samples, for which more observations have been generated as explained by the researchers posing the problem statement. The performance of the SVMs is bad, although they slightly outperform the NB when looking at the detectors as separate entities. When their predictions are combined, the scores are similar. The neural network approach at first seemed promising but was not able to deliver as was hoped for.

The main limitations that have required much time have been the difficulty to obtain reliable data, and hardware. The researchers state that more data needs to be simulated, but the documentation to do so is poor. For instance, signals can only be directly generated on top of Gaussian noise. If the noise configuration is set to be non-stationary, or contains gaps, the signal needs to be embedded a posteriori. For that, most generated data has Gaussian background, as opposed to the real data that needs to be predicted. Then, hardware has been a significant obstacle when training. *Kaggle* allows to run scripts in its servers, but there is a limitation in terms of HDD, CPU, GPU, and RAM. The latter has been the issue. Even though 30 GB are provided, the notebooks kept running out of memory regardless of how compressed the data was. This might be due to the fact that RAM adds up and does not get reduced immediately after objects are released. That is, *Kaggle* seems to cache lots of its objects and operations. Another slight limitation has been the complexity of the data, which requires a deep understanding of the physics involved, and bounds the capacity of extracting significant features.

Regarding the autoencoders, better data curation could have been applied to improve the variety of data that the model was trained on. As already mentioned, the data required expert knowledge to be generated and with poor documentation, it was not possible to properly generate a range of data that reflects the true observations. The amount of data was also limited, due to the sparsity thereof. Because of the hardware limitations mentioned earlier, it was often hard to explore different model architectures. Namely, when a model was incorrectly build, and RAM would overflow, a 30 minute wait of loading in all data followed. Merely getting things to fully work was already a major challenge which was overcome. It would have been interesting to explore transfer learning with the build in inception model from `Tensorflow` for example. Next to this, the assessment of the autoencoders was shallow, as it was mostly done on visual judgement. Again due to pooring in time in getting things to work in general, little time was left for exploring ways of assessing the performance of the autoencoder in a quantitative way.

In spite of these, much has been learnt throught the project.

References

[1] G2net detecting continuous gravitational waves. https://www.kaggle.com/competitions/g2net-detecting-continuous-gravitational-waves. Accessed: October 2022.

[2] Joe Bayley, Chris Messenger, and Graham Woan. Generalized application of the viterbi algorithm to searches for continuous gravitational-wave signals. *Phys. Rev. D*, 100:023006, Jul 2019.

[3] Joe Bayley, Chris Messenger, and Graham Woan. Robust machine learning algorithm to search for continuous gravitational waves. *Phys. Rev. D*, 102:083024, Oct 2020.

[4] LIGO. G2net: Realistic simulation of test noise. https://www.kaggle.com/code/vslaykovsky/g2net-realistic-simulation-of-test-noise. Accessed: December 2022.

[5] Vladimir Slaykovsky. Ligo document t0900149-v6. https://dcc.ligo.org/LIGO-T0900149/public. Accessed: December 2022.

[6] Rodrigo Tenorio, David Keitel, and Alicia M. Sintes. Search methods for continuous gravitational-wave signals from unknown sources in the advanced-detector era. *Universe*, 7(12), 2021.