Draft version April 19, 2019
Typeset using LaTeX **twocolumn** style in AASTeX62

Extending the reach of gravitational-wave detectors with machine learning

Tri Nguyen, Michael Coughlin, Rich Ormiston, Rana Adhikari

ABSTRACT

We apply long short-term memory (LSTM) neural networks as a time-series regression analysis technique to filter instrumental noises from gravitational-wave detectors at the Laser Interferometer Gravitational-Wave Observatory (LIGO). Unlike traditional neural networks, LSTM networks store and use information from their past inputs, thus robust in handling sequential data like gravitational-wave signals. Once trained, an LSTM network should be able to learn, predict, and subtract both the linear and non-linear noise coupling mechanisms, given there exist a set of witness channels monitoring these noises. The result would improve LIGO's sensitivity, most greatly at the low-frequency limit 20-100 Hz where noise features are expected to be easier to learn, and allow the detection of gravitational-wave sources currently below the noise floor.

1. INTRODUCTION

1.1. *Noise Regression Analysis at LIGO*

The Laser Interferometer Gravitational-Wave Observatory (LIGO) is the world's largest gravitational-wave observatory. Adopting a Michelson interferometer design, LIGO detects passing gravitational waves by measuring the induced differential arm length (DARM) between its two perpendicular 4-km arms (Adhikari 2004; Tiwari et al. 2015; The LIGO Scientific Collaboration 2015), LIGO has observed signals from stellar-mass black-hole and neutron star mergers (Abbott, B. P. et al 2016). However, the gravitational waves from many of these objects are still lying below the detector sensitivity limit. LIGO's noises consist of instrumental and environmental sources, each coupling to the DARM via a different, both linear and non-linear, mechanism. A detailed study of these mechanisms would allow us to filter out these noises, improve LIGO's sensitivity, and make gravitational-wave detections a more routine occurrence.

The current regression method at LIGO is based on the Wiener-Kolmogorov filter, which minimizes the squared error between the DARM channel and the predicted noises from the physical environmental monitor (PEM) channels (Tiwari et al. 2015; LIGO Scientific Collaboration 2007). The Wiener filter, however, fails to remove the non-linear contributions. In fact, characterizing and filtering out these non-linear noises are a challenging process, because the coupling mechanisms are often sophisticated. For example, two or more noise sources may interfere before coupling to the DARM.

1.2. *Neural Networks as a Noise-Filtering Technique*

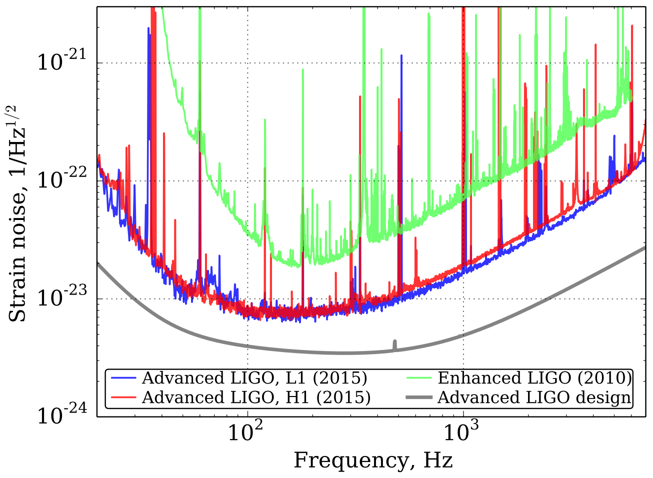


Figure 1. The sensitivity of LIGO Livingston (L1) and LIGO Hanford (H1) during the first observation run O1. Refer to Abbott et al. 2016.

Despite the complexity of the non-linear coupling mechanisms, a neural network may be able to learn them. Neural networks are a set of non-parametric, supervised machine learning algorithm often used in data classification and clustering. Modeled loosely after a human brain, a typical network may consist of up to a few thousands of nodes, each carrying parameters characterizing the features of the data. The network learns by looping over a labeled dataset (called the training set) and minimizing a set loss function via gradient descent. Once sufficiently trained, it is capable of recognizing highly complex patterns, such as language models, stock market, etc.

To filter LIGO's noise sources, we used long short-term memory (LSTM) networks, a special architecture which specializes in processing sequential inputs. A major strength of LSTM networks over traditional networks is their ability to store and use information from past inputs. As a result, LSTM networks are capable of taking into account the long-term dependencies in the data (Olah 2015). They are frequently used in speech recognition, grammar learning, and even DNA pattern recognition (Karpathy 2015). Given a sufficient amount of training data, an LSTM network should be able to learn, predict, and subtract both linear and non-linear coupling mechanisms, leading to an improvement in LIGO's sensitivity.

1.3. *Outline*

This paper is structured as follows. Section 2 discusses the objectives of the network, the metrics to measure its performance, and the data. Section 4 describes the procedures used to build an optimal analysis pipeline. Section 5 discusses the current progress and future plan of the project. Section 6 concludes the paper and discusses future plans.

2. OBJECTIVES

2.1. *Noise Coupling Mechanisms*

LIGO's noise sources can be categorized into fundamental and non-fundamental noises (The LIGO Scientific Collaboration 2015). Fundamental noises are imposed by quantum and statistical mechanics: these include photon shot noise, thermal noise, etc. Because fundamental noises define the baseline sensitivity limit, they can be reduced only by improving the detector design. On the other hand, non-fundamental noises consist of instrumental and environmental effects, such as seismic noise, magnetic noise, beam jitter, etc. They can be removed using statistical or machine learning techniques, given there exist witness channels monitoring the disturbance.

When a gravitational wave passes through the detector, the measured DARM is composed of the gravitational-wave signal and the noises. Consider the simple case of one witness channel w, the DARM output is:

$$h(t) = h_s(t) + \mathcal{T}[w(t_i)], \quad \forall t_i < t \tag{1}$$

where h_s is the signal and $\mathcal{T}$ is some (usually) non-linear function, called the transfer function, coupling the output of the witness channel into the DARM output. When there are multiple channels, the transfer function $\mathcal{T}$ takes in all of them.

The goal of the network is to predict the transfer function $\mathcal{T}$ of the non-fundamental noise sources and subsequently subtract these noises while keeping the gravitational-wave signals and the fundamental noises intact.

2.2. *Criteria for Success*

Trained on witness channels w_i at all time $t_i < t$, the network predicts a transfer function $\tilde{\mathcal{T}}$ and the DARM $\tilde{h}(t')$ at any later time $t_f >= t$, where

$$\tilde{h}(t_f) = \tilde{\mathcal{T}}[w_i(t_f)], \quad t_f > t \tag{2}$$

Here t is the cut-off time between the training and validation data. In general, the symbolic forms of the target $\mathcal{T}$ and the prediction $\tilde{\mathcal{T}}$ are not known. The network instead minimizes the mean square error (MSE) or the mean absolute error (MAE) between the true DARM h and the predicted DARM $\tilde{h}$.

Both the MSE and MAE are sensitive to data preprocessing, so they do not provide a meaningful absolute metric to evaluate the network's performance (although they are a meaningful relative metric to compare between different networks). The absolute performance of the network is evaluated by computing either the power spectral density (PSD) ratio or the coherence $c(f)$ between the DARM h and the clean DARM $h_c = h - \tilde{h}$. The coherence measures the correlation between these two channels at each frequency and is computed from their Fourier transforms. It is related to the ideal noise suppression factor $r(f)$ by:

$$r(f) = \frac{1}{\sqrt{1 - c^2(f)}}, \quad 0 \leq c(f) \leq 1 \forall f \tag{3}$$

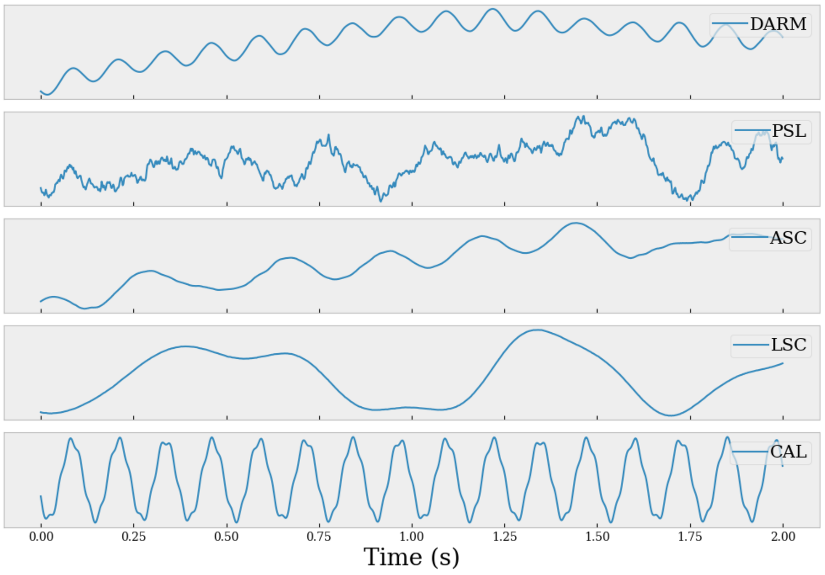
For example, if the coherence between h and h_c at some frequency is 0.9, the noise reduction factor at that frequency will be approximately 2.3 (M Coughlin 2014). We want to achieve a noise reduction factor of 2, corresponding to a coherence of about 0.87.

2.3. *Data*

We performed the analysis on both mock and real data. Mock data provide a reliable gauge to measure the network capacity, or its ability to learn complex functions. Mock data are generated by coupling white noises into the DARM output by a known, non-linear transfer function. For example, in our analysis, resonant noises are generated using the resonance function:

$$h(t) = \mathcal{F}^{-1}\left[\frac{\mathcal{F}[w(t)]}{w_0^2 - w^2 + i\frac{w_0 w}{Q}}\right] \tag{4}$$

where w_0 is the resonant angular frequency, and Q is the quality factor describing how underdamped the resonator is. Here $\mathcal{F}$ and $\mathcal{F}^{-1}$ denote the Fourier and inverse Fourier transform. Note though the coupling is in the frequency domain, the analysis is performed on the time domain. Mock data can also be generated from



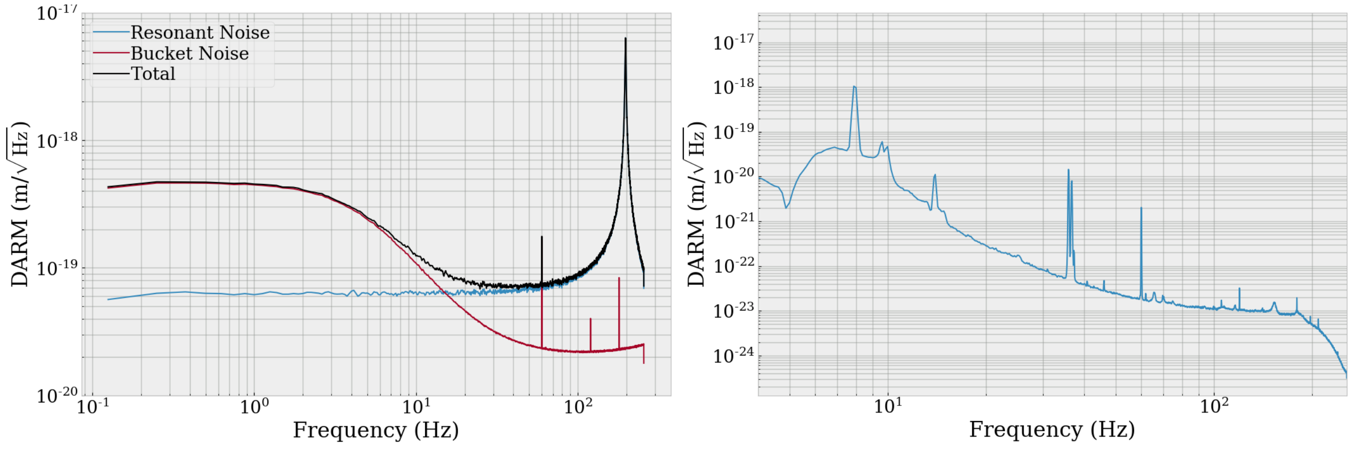


Figure 2. Top: Sample witness channels by subsystems from LIGO Hanford on August 14, 2017. Bottom: Mock resonant noise spectra (Left) with resonant frequency w_0 5 Hz and quality factor Q 100 and the LHO DARM spectra (Right) on the same date.

multiple channels. For example, bilinear data are generated by adding the product of the angular sensing noise and the beam spot motion noise to the DARM. To ensure the network removes only the targeted noises, we injected gravitational-wave signals and other background noises (called the bucket noises) into the mock, and later check if they are removed or distorted.

Ultimately, we would like the network to perform well on real data. While we do not know the true coupling mechanisms, we expect the mock data to be similar enough to real data and sufficiently capture their complexity. In addition, we may check if the network successfully subtracts known lines in the PSD. These include the calibration lines at 7 and 34 Hz, which are generated by moving the mirrors on-site periodically at these frequencies, and the 60 Hz line from the AC power grid of the United States.

The input data consists of multiple witness channels. Each witness channel has a duration of 2048 seconds and a sample rate of 512 Hz. The number of samples is 1,048,576. Figure 2 shows the time series of the witness channels of different subsystems in LIGO Hanford (LHO) on August 14, 2017. The subsystems are Pre-stabilized Laser (PSL), Alignment Sensing Control (ASC), Length Sensing Control (LSC) and Calibration (CAL). The PSDs of the resonance mock data and the real data are also shown.

3. NEURAL NETWORKS

3.1. *Traditional Neural Networks*

As briefly discussed in Section 1.2, neural networks are a set of machine learning algorithms designed to learn and perform a certain task, ranging from data classification to data regression. A typical network consists of feed-forward layers (i.e. the data stream moves through

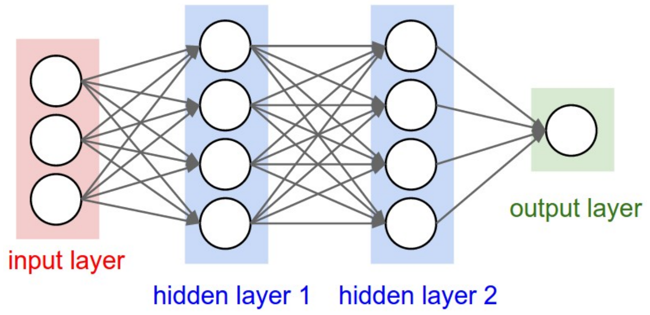


Figure 3. Top: The block digram shows a three-layer neural network. Refer to Karpathy 2017.

the network in only one direction). Each layer has multiple processing nodes, each connected to the nodes of the adjacent layers. Each node carries parameters (weights and biases) to characterize the data and an activation function to add some non-linearity to the output. Figure 3 shows a three-layer neural network (one input layer, two hidden layers, and one output layer). The layers are densely interconnected.

There are three steps of a network's training process: forward propagation, backward propagation, and gradient descent.

3.1.1. *Forward Propagation*

The data propagates forward (from the input layer to the output layer), and the network computes the output. The output of the layer l is computed via an affine (linear) transformation:

$$z^{[l]} = W^{[l]T} a^{[l-1]} + b^{[l]} \tag{5}$$

where $a^{[l-1]}$ is the output vector of the $(l-1)$th layer, $W^{[l]}$ is the weight matrix of shape $(n^{[l-1]} \times n^{[l]})$ and $b^{[l]}$ is the bias vector of shape $(n^{[l]} \times 1)$. Here $n^{[l]}$ is the number of hidden units of the lth layer. By convention, the input layer is the 0th layer. The output of the layer has the shape of $(n^{[l]} \times N)$ where N is the batch size, or the number of samples propagating through the network at once. If there is an activation function $f^{[l]}$, it is applied element-wise to the final output:

$$a^{[l]} = f^{[l]}(z^{[l]}) \tag{6}$$

Finally, the network computes a loss based on the output of the last layer. The MSE and MAE losses are:

$$mse = \frac{1}{N}\sum_{i}^{N}(y_i - \hat{y}_i)^2 \tag{7}$$

$$mae = \frac{1}{N}\sum_{i}^{N} y_i - \hat{y}_i \tag{8}$$

where y_i and $\hat{y}_i$ are the target and the prediction made by the network. Different losses may be used, as long as they effectively quantify the degree of error made by the network.

3.1.2. *Backward Propagation*

The loss propagates backward (from the output layer to input layer), and the network computes the gradients of the weights and biases. The gradients are computed analytically by applying the chain rule recursively. The backward propagation for the layer lth is:

$$\frac{\partial J}{\partial z^{[l]}} = \frac{\partial J}{\partial a^{[l]}} * f^{[l]'}(z^{[l]}) \tag{9}$$

$$\frac{\partial J}{\partial W^{[l]}} = \frac{1}{N}\frac{\partial J}{\partial z^{[l]}} a^{[l-1]T} \tag{10}$$

$$\left(\frac{\partial J}{\partial b^{[l]}}\right)_j = \frac{1}{N}\sum_{i}^{N}\left(\frac{\partial J}{\partial z^{[l]}}\right)_{ji} \tag{11}$$

$$\frac{\partial J}{\partial a^{[l-1]}} = W^{[l]}\frac{\partial J}{\partial z^{[l]}} \tag{12}$$

where $*$ denotes element-wise multiplication, and J is the loss. The derivative of J with respect to a matrix is a matrix of the same shape.

3.1.3. *Gradient Descent*

In a vanilla gradient descent algorithm, the weights and biases are updated as follows:

$$W_{i+1}^{[l]} = W_i^{[l]} - \alpha\frac{\partial J}{\partial W^{[l]}} \tag{13}$$

$$b_{i+1}^{[l]} = b_i^{[l]} - \alpha\frac{\partial J}{\partial b^{[l]}} \tag{14}$$

where α is the learning rate and J is the loss function. In addition, there exists more advanced gradient descent algorithms which perform better (e.g. RMS prop, gradient descent with momentum, etc.).

3.2. *LSTM Networks*

3.2.1. *Recurrent Neural Networks*

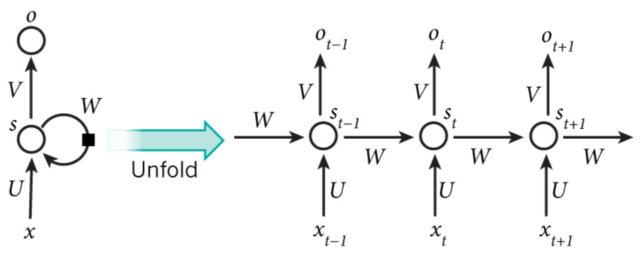


Figure 4. Unfold RNN structure. Refer to Britz 2015.

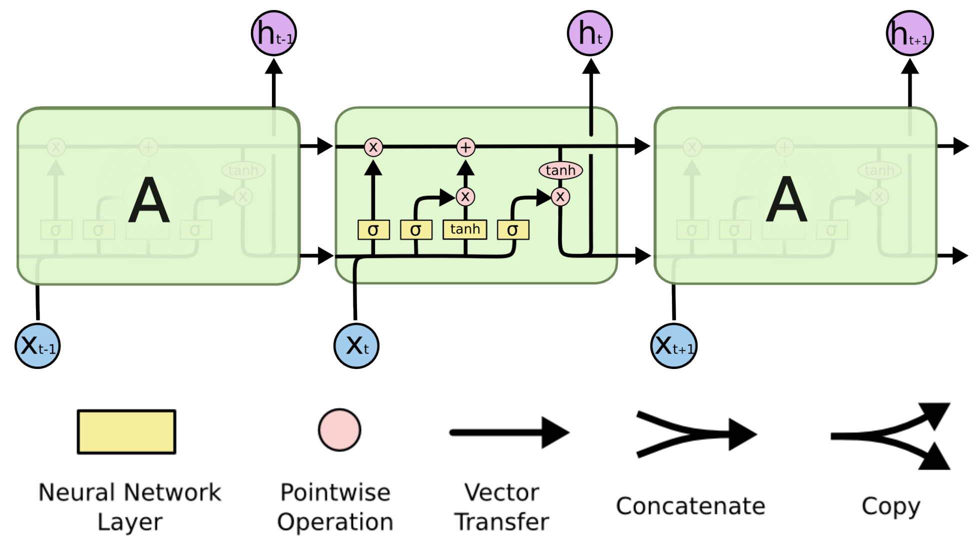


Figure 5. A typical LSTM layer contains four interacting sub-layers. Refer to Olah 2015.

LSTM networks are a special type of recurrent neural networks (RNNs), a subset of neural networks. RNNs have recursive loops such that they can store and use information from their most recent past inputs. Figure 4 shows the structure of a typical RNN. The memory of the network, called the hidden state, is represented by the horizontal line. It is thanks to this feature that RNNs are able to take into account the short-term dependencies of the data. The current input vector x_t and the previous state s_{t-1} are used to compute the current state by the equation:

$$s_t = f(Ux_t + Ws_{t-1}) \tag{15}$$

where f is an activation function and U and W are the weight matrices. The state s_{-1} required to compute the first state initialized to zeros. The network computes the output from the current state:

$$o_t = g(Vs_t) \tag{16}$$

where g is another activation function. The process is then repeated: the input x_{t+1} and state s_t are used to compute the state s_{t+1} and output o_{t+1}. During backward propagation, the weight matrices U, W, and V are updated via gradient descent.

3.2.2. *Advantages and Structures of LSTMs*

RNNs suffer from what known as the vanishing gradient problem. Vanishing gradient occurs when the gradients of a network decrease exponentially after each iteration of backward propagation. This prevents the weights from changing their values (as can be seen from Equation 13) and effectively stops the training process. One of the causes of the vanishing gradient is the activation function. If the derivative of the activation function is less than 1, vanishing gradient occurs. On the other hand, if the derivative is greater than 1, the gradient explodes, meaning it increases exponentially after each iteration. For example, the derivative of $\tanh(x)$ (which is $1 - \tanh^2(x)$) has a range of $(0, 1]$.

LSTMs solve the vanishing gradient problem by introducing the cell state, which is represented by the horizontal line in Figure 5. Like the hidden state, the cell state passes long past information. However, the activation function here is the identity function, so the derivative is always 1. This effectively solves the vanishing gradient problem.

LSTMs also introduce gating. A typical LSTM unit has 3 gates. The forget gate (the first horizontal line in Figure 5) decides which information to keep in the cell state:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{17}$$

where σ is the sigmoid function, W_f and b_f are the weight and bias. Here $[h_{t-1}, x_t]$ means the previous output vector h_{t-1} is concatenated with the current input vector x_t. The input gate (the second and third horizontal lines) adds new information to the cell state:

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{18}$$
$$\tilde{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C) \tag{19}$$

The cell state is updated as follows:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{20}$$

where $*$ denotes element-wise multiplication. Finally, the output gate computes the output from the current input and the cell state:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{21}$$
$$h_t = o_t * \tanh(C_t) \tag{22}$$

For more details of LSTMs, refer to Olah 2015.

4. ANALYSIS

Building the optimal neural network requires carefully choosing the best architecture (e.g. the number of hidden layers, activation function, etc.) and hyperparameters. Hyperparameters are those govern the learning process (e.g. learning rate, regularization strength, etc.). They cannot be learned from the data, and different problems require different sets of hyperparameters.

Finding the optimal hyperparameters is an empirical process. In other words, it requires experimenting with different parameters and evaluating their relative performance. In our analysis, we started by picking a simple architecture (so training will go faster) and increasing the complexity as needed. Initially, we wanted the network to overfit a small training dataset to make sure it is capable of learning the transfer function. Then, we reduced overfitting by choosing the optimal hyperparameters via a machine learning process called hyperparameter tuning, or hyperparameter optimization, which will be explained later in more details.

4.1. *Data Preprocessing*

Data preprocessing is one of the most important machine learning processes. It often involves reformatting, cleaning (e.g. removing and/or fixing missing data), resampling and transforming data. In a neural network, data preprocessing helps speed up the learning process, reduces overfitting, and fixes the vanishing/exploding gradient problem in backpropagation.

In our analysis, we down-sample or up-sample the channels so they have a sample rate of 512 Hz. Then, we apply standard scaling and lookback windows.

4.1.1. *Standard Scaling*

This is the most common form of preprocessing. For each witness channel, we normalize the mean and standard deviation to 0 and 1. Mathematically speaking, for the witness channel j and the data point i is transformed as follows:

$$x_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j}$$

where $\mu_j = \frac{1}{N}\sum_k x_{kj}$ and $\sigma_j = \frac{1}{N}\sum_k (x_{kj} - \mu_j)^2$ are the mean and standard deviation over the samples in the witness channel. Here N is the number of samples, which is 1,048,576 as mentioned in Section 2.3.

4.1.2. *Lookback Windows*

Because the length of each channel is large, the network cannot fit all the data into backpropagation memory. Furthermore, the number of parameters is so great that the learning process becomes impossible and computationally expensive. As a solution, we divide each channel into many short and overlapping time series, each consisting of 16 samples. As a result, our dataset has 1,048,546 data examples. In each example, each witness channel is a time series of length 16.

4.2. *Picking the Network Architecture*

4.2.1. *Current architecture*

The current analysis pipeline consists of five separate LSTM networks. Each network uses a different set of hyperparameters and gradient descent algorithms, and tunes on a different frequency band. The frequency cutoffs are respectively 3-9 Hz, 10-13 Hz, 20-30 Hz, 30-41 Hz, and 57-63 Hz. We employ multiple small networks across multiple frequency bands instead of one big network because they allow us to easily inject and test out new noise sources. For example, injecting a source at 5 Hz only requires us to re-train the first network (3-9 Hz), not the entire pipeline. In addition, each small network is significantly easier and faster to train. Because the networks are trained separately, we may further speed up training via parallel computing on a computer cluster, a process that cannot be done easily if we use one big network.

Each network contains multiple LSTM layers followed by multiple fully-connected (Dense) layers. Between each layer (except for the output layer) is a Batch Normalization layer, which normalizes the mean and standard deviation across each channel to 0 and 1. We found batch normalization helps reduce the vanishing gradient problem and makes the network less sensitive to initialization. Each LSTM layer returns a sequence which gets fed into the next LSTM layer (with the exception of the last LSTM layer, which returns a vector to the first Dense layer). The full network structure in the case of one witness channel is summarized in Table 1. In total, there are 5869 trainable parameters (weights and biases). All networks use the Adaptive Moment Estimation, or Adam, gradient descent algorithm (Kingma & Ba 2014). The biases are initialized as ones, and the initial weights are sampled from a uniform distribution within $(-\sigma, \sigma)$, where

$$\sigma = \sqrt{\frac{6}{n_{in} + n_{out}}}$$

Here n_{in} and n_{out} are the number of features of the input and output vector respectively. This initializer is called the Xavier uniform initializer (Glorot & Bengio 2010). Alternatively, the initial weights may be sampled from a Gaussian distribution with mean 0 and standard deviation σ. This is known as the Xavier normal initializer.

Layers	Output Shape	# Params
LSTM 1	(N, 16, 16)	1152
Batch Norm 1	(N, 16, 16)	64
LSTM 2	(N, 16, 8)	800
Batch Norm 2	(N, 16, 8)	32
LSTM 3	(N, 16, 8)	544
Batch Norm 3	(N, 16, 8)	32
LSTM 4	(N, 16, 8)	544
Batch Norm 4	(N, 16, 8)	32
LSTM 5	(N, 16, 8)	544
Batch Norm 5	(N, 16, 8)	32
LSTM 6	(N, 16, 8)	544
Batch Norm 6	(N, 16, 8)	32
LSTM 7	(N, 16, 8)	544
Batch Norm 7	(N, 16, 8)	32
LSTM 8	(N, 6)	360
Batch Norm 8	(N, 6)	32
Dense 1	(N, 8)	56
Batch Norm 9	(N, 8)	32
Dense 2	(N, 8)	72
Batch Norm 10	(N, 8)	32
Dense 3	(N, 8)	72
Batch Norm 11	(N, 8)	32
Dense 4	(N, 8)	72
Batch Norm 12	(N, 8)	32
Dense 5	(N, 8)	72
Batch Norm 13	(N, 8)	32
Dense 6	(N, 8)	72
Batch Norm 14	(N, 8)	32
Dense 7	(N, 8)	72
Batch Norm 15	(N, 8)	32
Dense 8	(N, 8)	72
Batch Norm 16	(N, 8)	32
Dense 9	(N, 1)	9

Table 1. Current architecture of each LSTM network, in the case of one witness channel. Here N is the batch size. The lookback window size is 16.

4.2.2. *The Importance of Batch Normalization*

The initial architecture of the networks did not have batch normalization layers. As a result, the networks were highly sensitive to the initial weights: given the same dataset and hyperparameters, they might give completely different loss value in two different runs. The loss distribution after 10 epochs was bimodal, as shown in Figure 6. As evidenced by the training and testing loss curve in Figure 7, once the networks started to get out of a local minimum or saddle point, the gradient descent would always converge. This further shows that the networks were highly sensitive to the initial weights. Approximately 1/5 of the networks failed to converge within 10 epochs.

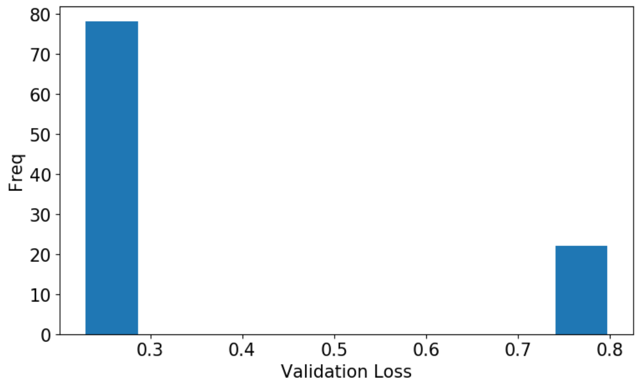


Figure 6. Validation loss distribution of 100 identical runs is bimodal. Approximately 1/5 of the network fails to converge within 10 epochs.

Although the underlying problem remains unclear, we suspect the networks ran into a vanishing gradient problem as the architecture consists of a large number of LSTM and Dense layers. Adding a batch normalization layer after each LSTM/Dense layer eliminated the problem. Once the input of each layer was normalized, the data became easier to learn because the weights could no longer reach the extreme, too small or too high, values (which would result in extreme values of the gradient). Even better, a positive side effect of batch normalization was the training process takes much fewer epochs to converge (see Figure 7).

4.3. *Hyperparameter Tuning*

We apply the holdout cross-validation method. In particular, the data are partitioned into a training set and a test set. After each iteration through the training set, the network runs on the test set. As discussed in Section 2.2, the test performance is evaluated by computing the MSE or MAE between the true DARM and the predicted DARM. This provides an insight on how the network performs on an independent, unknown dataset. We choose the hyperparameters that result in the lowest MSE or MAE. In our analysis, we divide the time series into two equal subsets, with the training set being the first half. The size of the training and test set is 524,288 (as described in Section 2.3), which should be sufficient to capture all data features.

To search for the optimal hyperparameters, we may apply three different algorithms: random search, Gaussian process regression, and tree-structured Parzen estimator.

4.3.1. *Random Search*

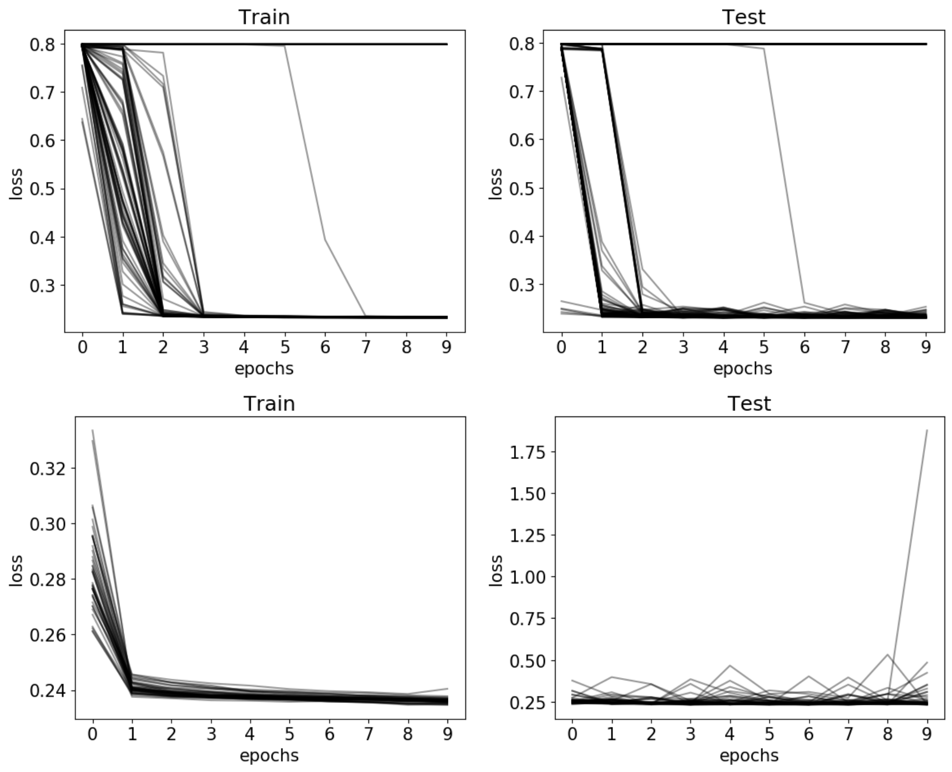


Figure 7. The training loss (left column) and validation loss (right column) per epoch of 100 runs, before (top) and after (bottom) adding batch normalization.

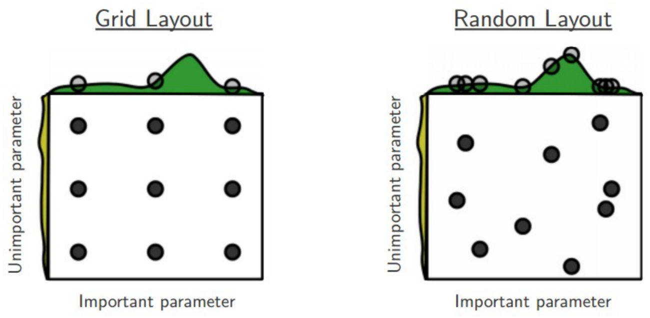


Figure 8. Random search explores the parameter space more efficient than grid search (Bergstra & Bengio 2012).

As its name suggested, random search involves randomly sampling the hyperparameters given some prior guesses on them. Scale parameters (e.g. learning rate, learning rate decay, regularization strength, etc.) are sampled from a Jeffreys prior, and location parameters (e.g. dropout, recurrent dropout, etc.) are sampled from a uniform prior. In cases where we also tune discrete parameters like the activation function, the parameter will be chosen with equal probability from a list of possible choices.

At first, the algorithm seems counterintuitive. Why not pick a list of possible value for each parameter and systematically test out all possible combinations? This method is known as the grid search; it has been shown to be much less efficient in scanning the parameter space. This is because some parameters may greatly affect the cost function, while others only have a second-order effect. Given the same amount of trials, random search allows the exploration of more values for each parameter (see Figure 8).

Random search is a simple algorithm for hyperparameter tuning. However, there exist more advanced algorithms that employ probability theories to predict the mapping between the parameters and the validation loss. In general, these algorithms decide a set of parameters the network should try based on previous observations. In this analysis, we use the Gaussian process regression and the tree-structured Parzen Estimator.

4.3.2. *Gaussian Process Regression*

A Gaussian process can be thought as a distribution over stochastic (random) functions. For each value of x in the parameter space, the Gaussian process regression assumes the loss function $y = f(x)$, which is a stochastic function whose noises follow a Gaussian distribution with mean μ and standard deviation σ. This is a reasonable assumption for the loss of a neural network because of the many random processes involved in training (e.g. dropout, initialization, etc.). In the case of multiple parameters, the noises follow a multivariate Gaussian dis-

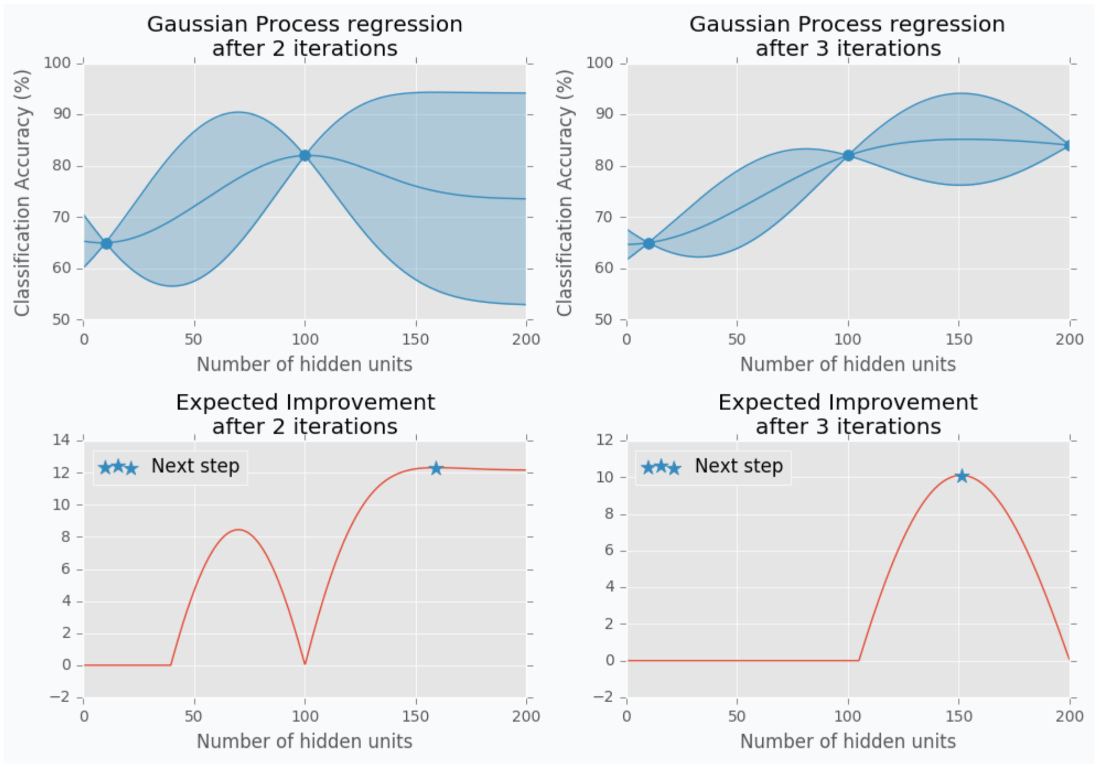


Figure 9. Gaussian process regression for hyperparameter tuning. In this example, the algorithm predicts the number of hidden units that will maximize the validation accuracy of a Dense neural network. Refer to Shevchuk 2016.

tribution. Figure 9 shows an example of the Gaussian process regression for hyperparameter tuning.

Given a prior function distribution and some past evaluations (hyperparameters and losses), the algorithm predicts the loss for each point in the parameter space by constructing the posterior distribution[1]. The network is then tested on the point with the minimum expected loss, to within about 2-3 standard deviations of the mean.

The greatest advantage of the Gaussian process regression is the ability to account for the uncertainty of the calculation and the relation between each hyperparameter. However, the algorithm does have some important drawbacks. For example, it is tricky to incorporate categorical parameters (e.g. activation function, initializer, etc.) into the prediction. It is also difficult to select the right hyperparameters for the Gaussian process, such as the prior function distribution. Lastly, when the number of parameters exceeds a few dozens, the algorithm becomes computationally expensive (Shevchuk 2016).

4.3.3. *Tree-structured Parzen Estimator*

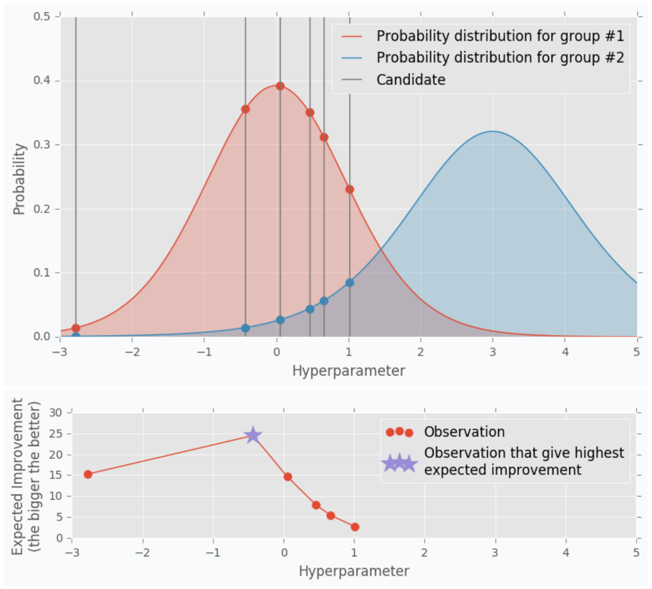


Figure 10. TPE predicts the best parameters by constructing a likelihood ratio of the two groups. Refer to Shevchuk 2016.

In the tree-structured Parzen estimator (TPE), the collected observations are divided into two groups: one with the best validation performance and one with all the others. The algorithm picks the parameter that is most likely belong to the first group and less likely be-

[1] For a detailed mathematical calculation, please refer to Do & Lee 2008

long to the second group. It does so by constructing the likelihood probability (unlike the Gaussian process, which constructs the posterior probability). In particular, TPE picks the parameter x with the highest expected improvement:

$$r(x) = \frac{l(x|D)}{g(x|D)}$$

where $l(x|D)$ and $g(x|D)$ are the likelihoods of x belong to first and second groups respectively. Figure 10 shows how TPE predicts the next parameters the network should try.

In general, TPE is less sensitive to random noise than the Gaussian process is. It also has fewer hyperparameters and allows the prediction for categorical parameters more easily. However, the biggest disadvantage of TPE is that it cannot take into account the relation between parameters because it selects the parameters independently.

4.3.4. *Hyperparameter Space*

Table 2 summarizes our hyperparameter space. In our analysis, we evaluated the network on about 100-200 sets of parameters. We used both random search and TPE.

Parameters	Distribution	Values
Activation	Categorical	Tanh, Relu
Bias initializer	Categorical	Zeros, Ones
Weight initializer	Categorical	Xavier uniform, normal
Learning rate	Log-uniform	$x \in (10^{-6}, 10^{-2})$
Learning rate decay	Log-uniform	$x \in (10^{-8}, 10^{-2})$
Dropout	Uniform	$x \in (0.1, 0.9)$

Table 2. Hyperparameter space.

4.4. *Selecting Witness Channels: Ablation Study*

Not all witness channels carry the same weight. Some are more important for the subtraction, while some only have a second-order effect. Furthermore, some channels may share the same, correlated information (degeneracy). Currently, the data include a selected few channels. Ultimately, we would like to understand which channels are important and should be included. We cannot simply use all channels because the networks will become too hard to train (too many features). More importantly, training will become too expensive in both running time and memory. Despite the lack of understanding of the underlying physics, we can still study the importance of each channel via an ablation study.

In machine learning, an ablation study is a process of removing some features of the data and seeing how it affects the performance. After the networks were trained, we set a witness channel in the validation data to zero and compared the subtraction before and after. Should the networks fail, we concluded that the witness channel was important for the subtraction on the frequency band. We repeated the procedure for every channel.

It is possible to use this study and try to infer some detector physics. For example, if the networks fail to remove a calibration line after we remove a CAL channel, the channel provides coupling information of the line. However, this should always be done with caution. If the networks overfit the training data and pick up some of its idiosyncratic features, the same inference might not be reliable.

4.5. *Machine Learning Framework*

Our current machine learning framework for noise regression is DeepClean[2]. DeepClean is written on top of Keras, a high-level Python neural networks API which allows fast and flexible creation and development of neural network models (including Dense, LSTM, Convolution Net, etc.). Keras is a Google Brain's TensorFlow backend. It is able to run across multiple platforms (CPUs, GPUs, TPUs) and thus allows us to speed up training by parallelizing our calculation across GPU or TPU cores.

DeepClean is capable of creating multiple networks, each with a different architecture, to subtract noises across multiple frequency bands. After preprocessing the data using the methods in 4.1, DeepClean trains and tests each network separately. On each run, it loops over each frequency band and uses the assigned network to perform the subtraction; the final subtraction is a combined result from all networks. Figure 11 shows an example DeepClean output, in which it subtracts the beam jitter noise from LIGO's first observational run O1.

DeepClean can also apply hyperparameter tuning using the algorithms described in section 4.3. Each network is tuned separately. Note that the Gaussian process in DeepClean hyperparameter tuning is a 1-dimensional Gaussian process, which is not ideal because it eliminates the algorithm's greatest advantage over TPE: the ability to take into account the relation between parameters. A multivariate Gaussian process tuning is currently being developed.

5. RESULTS

[2] Available at https://git.ligo.org/rich.ormiston/DeepClean

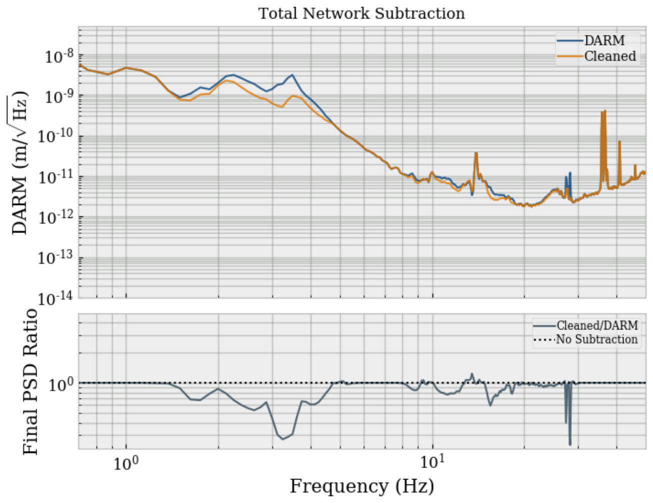


Figure 11. In this example of DeepClean output, the networks subtract the beam jitter noise from LIGO's first observational run O1.

5.1. *LIGO Hanford Data*

Figure 13 shows the DARM spectral density on August 14, 2017, and the subtraction in the frequency bands 3-9 Hz and 57-63 Hz before and after hyperparameter tuning. In Figure 13, these frequency bands are highlighted because they include the calibration line at 7 Hz and the AC power line at 60 Hz respectively. In the 3-9 Hz band, tuning results in a better subtraction of the calibration line. However, this was not the case for the subtraction of the AC frequency line in the 57-63 Hz band, where the un-tuned network gave a slightly better performance. Furthermore, both networks overfit the training data. The un-tuned network used the default adam (the gradient descent algorithm) parameters, which were known among the machine learning community to perform well on many problems. In addition, because the parameter space was large (see Table 2, searching over only 100 sets of parameters might not be sufficient. Figure 12 shows the total subtraction before and after tuning. Note that the frequency band of the intermediate network (Loop_1) was changed from 10-14 Hz to 10-13 Hz in the latter case to avoid collision with a spectra line at about 14 Hz (which causes the subtraction to add noises to the DARM).

5.2. *Resonant Mock Data*

Figure 14 shows the subtraction in the 10-250 Hz frequency band and the expected outcome. The resonant frequency was at about 13.15 Hz, and the quality factor was 1000. The network successfully subtracted the resonant noises and left out the 60-Hz AC power line and the rest of the bucket noise, which were not supposed to be removed. There was no tuning result on

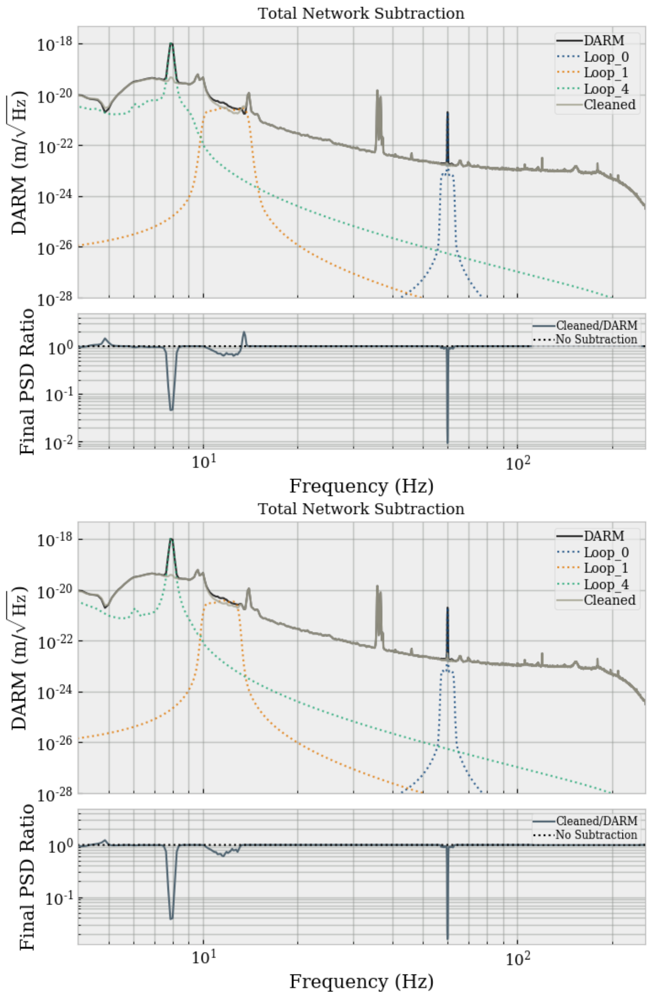


Figure 12. The DARM spectral density on August 14, 2017 and total subtraction before (top) and after (bottom) hyperparameter tuning.

this data. Note the normalization of the resonant noise matched the scaling of the bucket noise (to within 1-2 orders of magnitude), so neither the resonant noise nor the bucket noise dominated the spectra. If these cases occur, the networks will fail to produce any meaningful subtraction.

5.3. *Selecting Witness Channel: Ablation Study*

Figure 15 shows the result of the ablation study on the H1 August data in the 10-15 Hz frequency band. Here, the witness channel "ASC-CHARD_Y_OUT_DQ" was set to zero, and the subtraction fails. This shows that this alignment and sensing control channel contributes somewhat to the subtraction power of the network on this frequency band.

6. CONCLUSION

DeepClean uses a series of LSTM networks to predict LIGO's noise coupling mechanisms and subtract them.

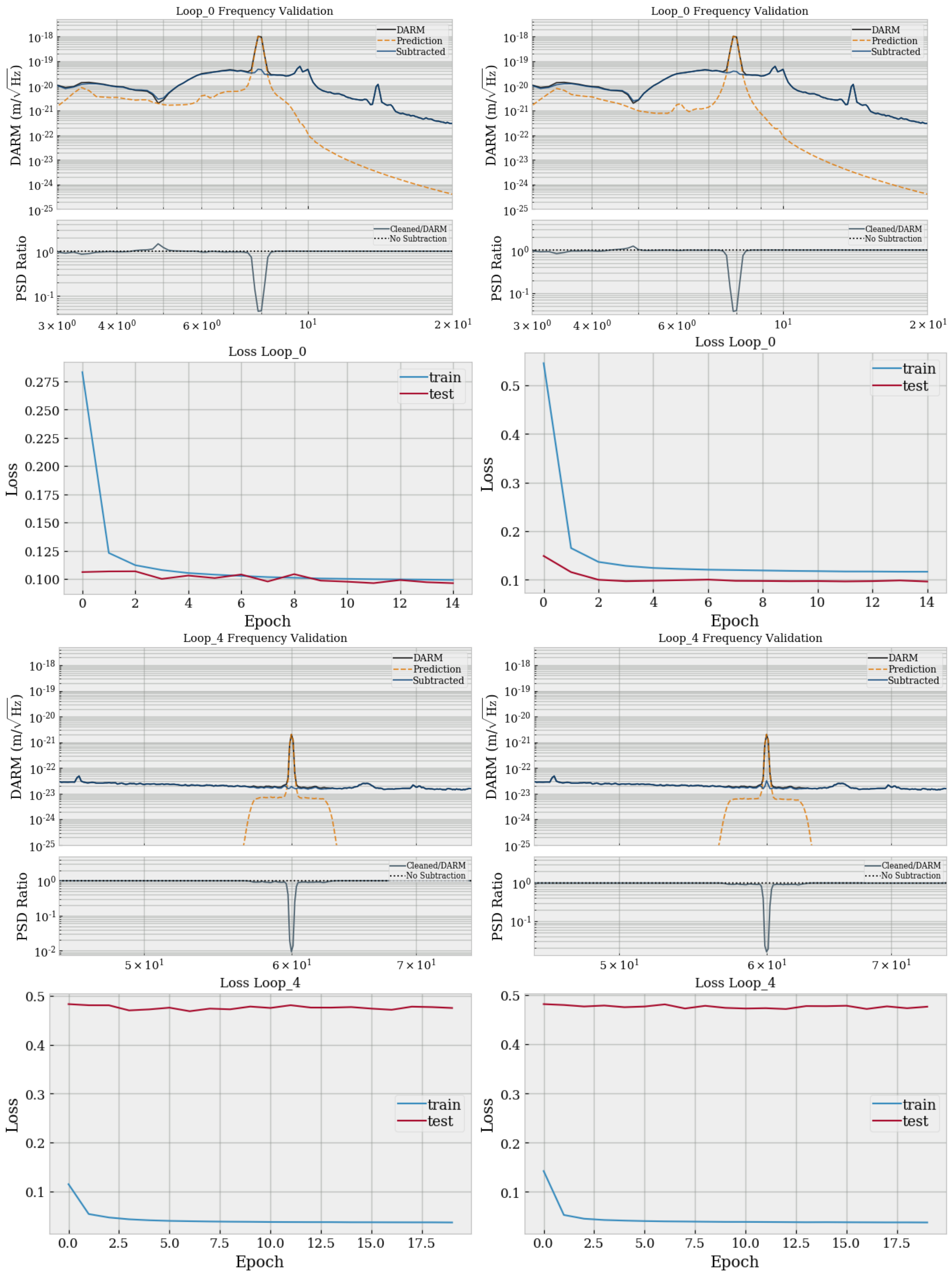


Figure 13. Top: The DARM spectral density on August 14, 2017 before (left) and after (right) hyperparameter tuning. The frequency band is 3-9 Hz. In this case, hyperparameter tuning results in a better subtraction of the calibration line at 7 Hz. Bottom: The DARM spectral density on August 14, 2017 before (left) and after (right) hyperparameter tuning. The frequency band is 57-63 Hz. In this case, hyperparameter tuning does not result in a better subtraction of the AC power line at 60 Hz.

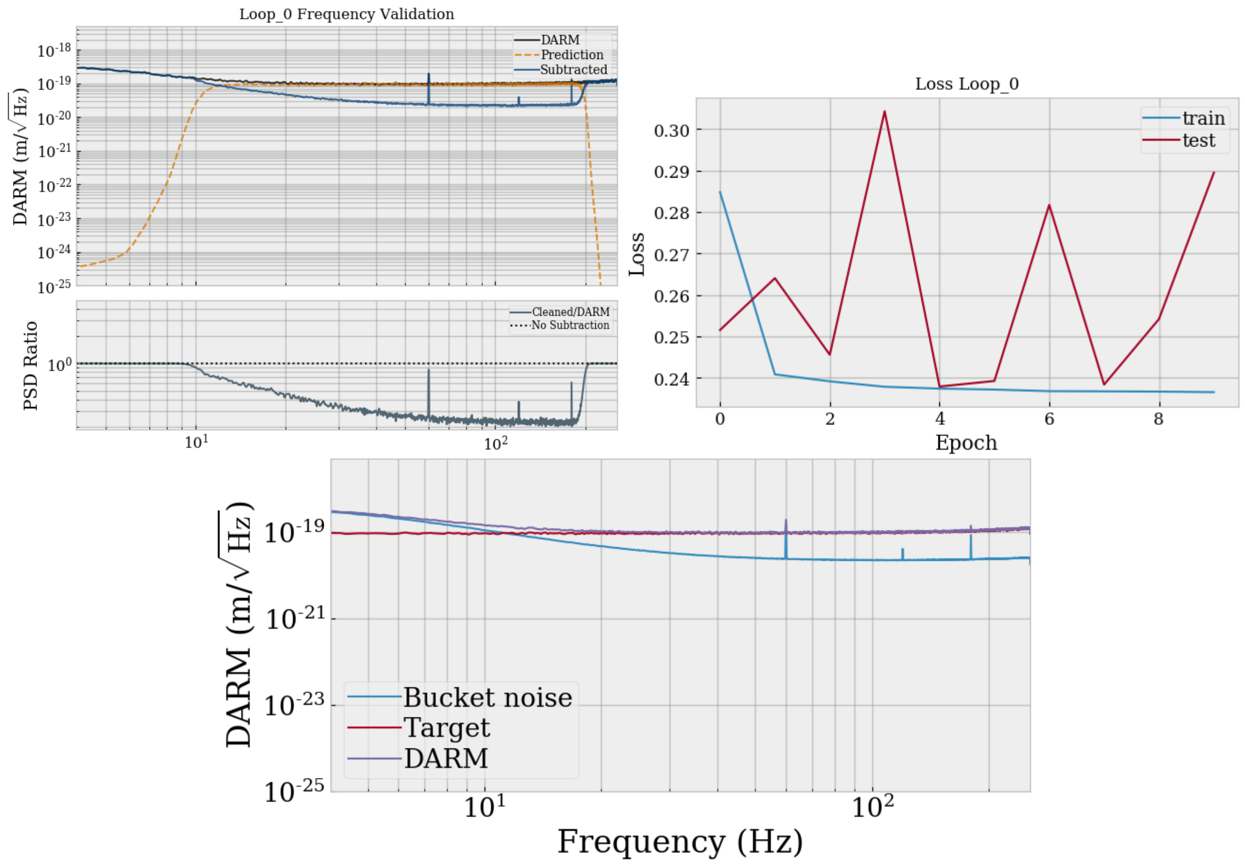


Figure 14. Top: The subtraction in the 10-250 Hz band (left) and the loss curve (right). The subtraction has successfully left out the bucket noise. Bottom: The expected outcome. The spectra is dominated by the resonant white noise. The resonant frequency is about 13.15 Hz, and the quality factor is 1000.

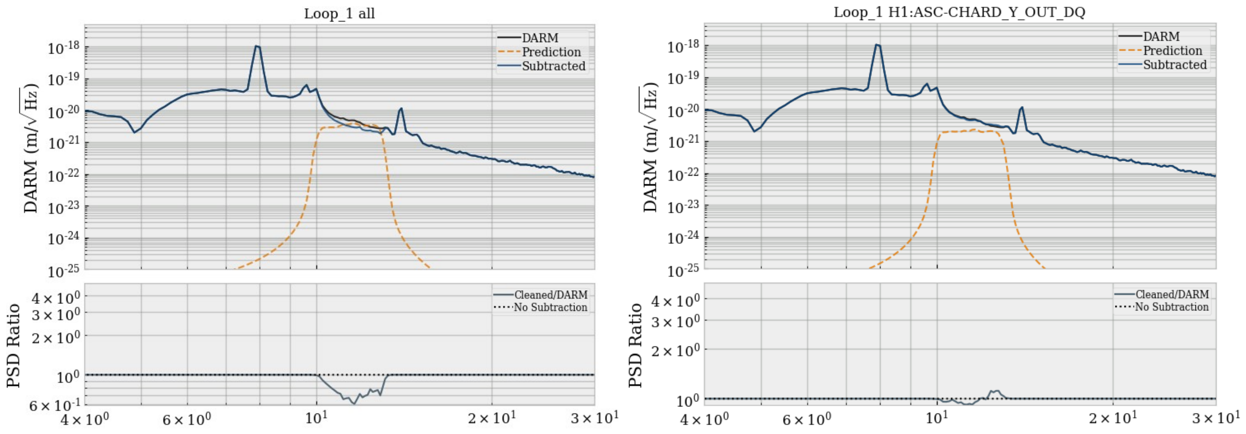


Figure 15. The subtraction result on the H1 August data in the 10-15 Hz frequency band before (left) and after (right) removing the channel “ASC-CHARD_Y_OUT_DQ”.

It is a powerful tool to detect weaker gravitational-wave signals and to understand the physics of the coupling mechanisms. Our goal is to apply DeepClean to data from the first observation (O1) and to further real-time detection (i.e. analyze an hour of data in less than an hour). We expect better subtraction at the 20-100 Hz frequency band (as the noise features are easier to learn). Because the frequency of most compact binary coalescences is within this band, if succeeded we may be able to discover more objects and/or uncover new information on past detections.

However, further study and optimization are needed before DeepClean can be applied for these purposes. The first step is to understand which witness channels are important for each bandwidth and should be included in the analysis. We may do so with a correlation study (as an alternative to the ablation study mentioned) by, for example, computing the coherence between the channels. Moreover, we suggest testing the pipeline with different network architectures. This includes but are not limited to applying new architectures, such as the 1-dimensional convolutional neural networks known for its image processing power, and using different loss functions. Currently, we are experimenting with the average spectral density (ASD) loss function, which is defined as the mean of the absolute value of the Fourier transform of the difference between the true DARM and the predicted DARM.

REFERENCES

Abbott, B. P., et al. 2016, Phys. Rev., D93, 112004, [Addendum: Phys. Rev.D97,no.5,059901(2018)]

Abbott, B. P. et al. 2016, Phys. Rev. Lett., 116, 061102

Adhikari, R. 2004, PhD thesis, Massachusetts Institute of Technology, Massachusetts, USA

Bergstra, J., & Bengio, Y. 2012, JMLR, 13

Britz, D. 2015, Recurrent Neural Networks Tutorial, Part 1 Introduction to RNNs

Do, C. B., & Lee, H. 2008, Lecture notes on Gaussian processes, Stanford CS229

Glorot, X., & Bengio, Y. 2010, in Proceedings of Machine Learning Research, Vol. 9, Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, ed. Y. W. Teh & M. Titterington (Chia Laguna Resort, Sardinia, Italy: PMLR), 249–256

Karpathy, A. 2015, The Unreasonable Effectiveness of Recurrent Neural Networks

—. 2017, Neural Networks Part 1: Setting up the Architecture, Stanford CS231n

Kingma, D. P., & Ba, J. 2014, CoRR, abs/1412.6980, arXiv:1412.6980

LIGO Scientific Collaboration. 2007, arXiv, 0711.3041

M Coughlin, J Harms, e. a. 2014, Classical and Quantum Gravity, 31, 215003

Olah, C. 2015, Understanding LSTM Networks

Shevchuk, Y. 2016, Hyperparameter optimization for Neural Networks

The LIGO Scientific Collaboration. 2015, Classical and Quantum Gravity, 32, 074001

Tiwari, V., et al. 2015, Class. Quant. Grav., 32, 165014