Fast Radio Burst Detection With A Convolutional Neural Network

Ben Cravens

October 26, 2019

1 Background

1.1 What is a Convolutional Neural Network?

Neural networks are biologically inspired algorithms that can be modeled as directed, weighted graphs. These algorithms learn functions from input data, and can theoretically be trained to approximate any function. In this assignment, I trained a neural network to detect the decay rate and oscillation frequency of a simulated noisy radio burst.

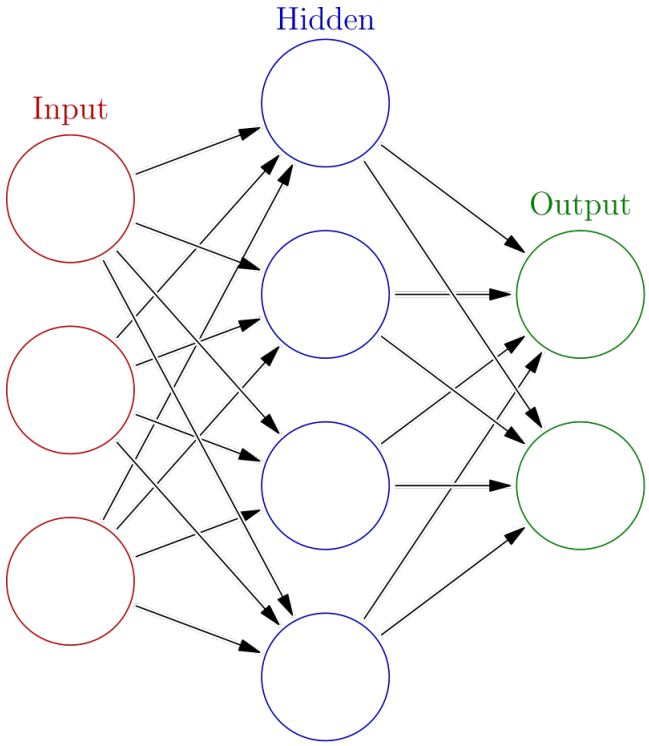


Figure 1: Image taken from `https://tinyurl.com/nfhgnkv`

This is a diagram of the basic structure of a neural network. The network consists of an input layer, which takes in external data such as images or numerical values, and operates on it in some way. These inputs are then

passed to the next layer, a "hidden layer", according to the weights of the connections between the input layer and the hidden layer. In most models, the layers are fully connected, meaning that every node in the layer connects to every node in the next layer.

The individual nodes in each layer are known as neurons, and they behave in a similar way. The neurons in the "hidden layers" are each outfitted with an activation function, which defines the output of the node or neuron given a set input. In this way, the information from the input is first operated upon by the input layer, often in a way that simplifies and cleans it, and it is then passed through the hidden layers to be processed by the activation functions, reshaped at the end by the last layer to produce a coherent output.

The sequence of layers in the network can be modeled by a loss function. This loss function tells us how "well" the neural network is doing at its task. This loss function forms a high-dimensional "loss landscape", and the goal of training the neural net is to find the lowest possible point in this landscape, in other words a global minima. To find this point, most neural networks use a gradient method such as gradient descent or conjugate gradient. These are numerical methods that find the minima of a function by taking the information about its gradient at a given point (which as we learned in multi-variable calculus tells us the direction of steepest descent). To find the gradient of the loss function, an algorithm called back-propagation is used, which "propagates" the derivative in the error with respect to each set of weights using the chain rule.

The specific architecture used in this assignment is a convolutional neural network. This type of neural network is commonly used in signal processing, such as time series analysis (1d), or more commonly in image processing (2d). This network architecture has been around since the 1980s, but development in applications of CNN's has exploded in recent years as computers become more powerful and the quality and quantity of labeled data increases.

The two main layer types of a CNN are convolutional layers and pooling layers. The convolutional layers simply perform the correlation operation on their inputs. When the layer is trained well, this acts to extract features from the image, as the neurons in the layer arrange themselves into feature maps, which correspond to things one wishes to identify in the image as specified

by the loss function. The pooling layers reduce the resolution of the image to increase the in variance of the neural network to translation, scaling and skewing of features it is trying to identify.

1.2 What is categorical cross entropy?

In a neural network, there is something called a loss function. This is a function that maps values calculated from the neural network onto a real number which represents a cost or an error. The neural network minimizes this "loss" function by tuning its parameters or weights so that it can perform better on the task it is being trained to do.

In the case of the simulated fast radio burst detection assignment, the original code given to me classified the frequency of the radio burst into different bins, each corresponding to a certain frequency range. The output of the neural network is thus a probability distribution spread across different frequency ranges, where the true frequency of the signal lies within one bin. Categorical cross-entropy is a loss function that works well to score the neural network in this situation, as it compares the predicted frequency distribution to the actual distribution, and then returns an error or loss.(*Categorical Cross-entropy* n.d.)

Categorical cross-entropy is defined as such:

$$L(y, x) = -\Sigma_{j=0}^{M}\Sigma_{i=0}^{N}(y_{ij}log(x_{ij})) \tag{1}$$

Where here y is the true distribution, and x is the predicted distribution.

2 Implementation

2.1 Model architecture

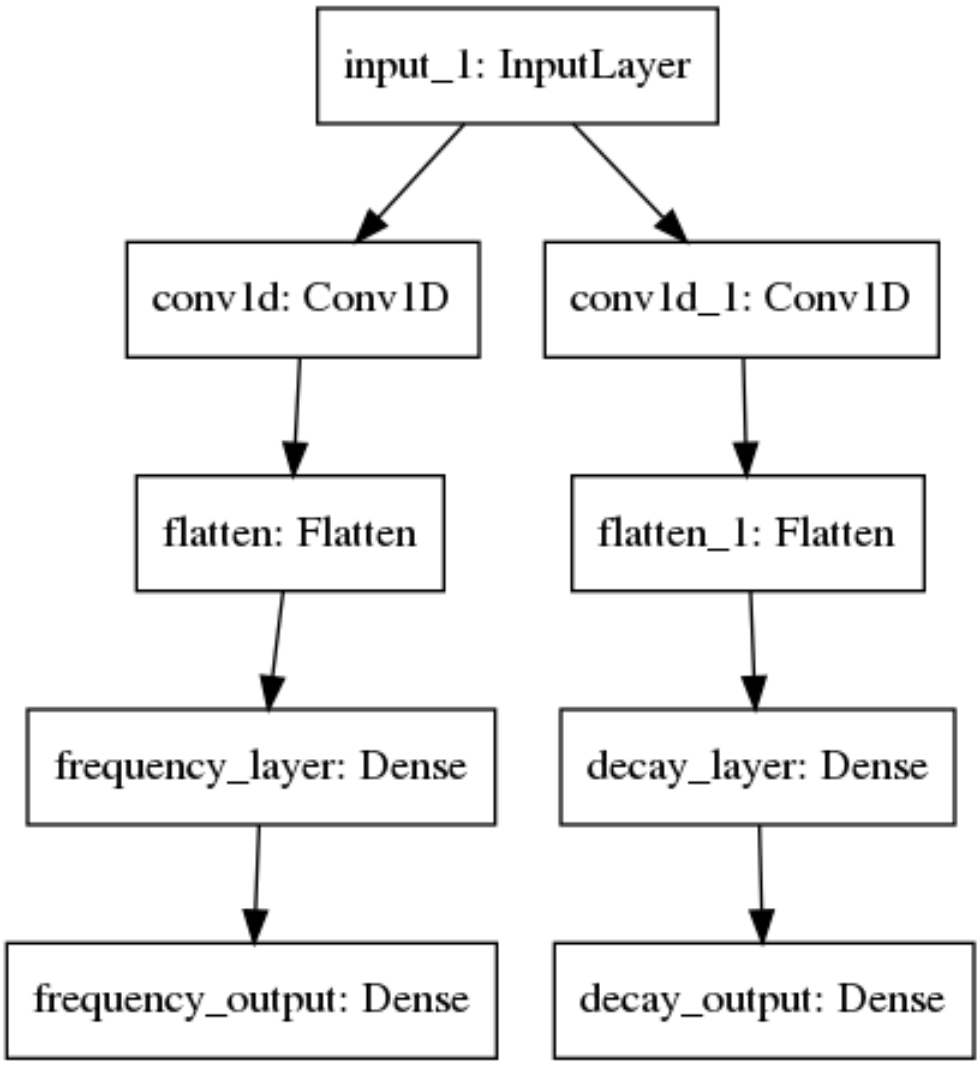


Figure 2: Output of tensor graph

I chose to eschew the original classification architecture in favour of a regression approach. This means that instead of the net being trained to assign a probability that the decay and frequency values of the signal are within certain ranges, the network outputs specific values for both decay and frequency. This meant that for the loss function, I used a distance metric instead of the aforementioned categorical cross-entropy. The metric I used was mean-squared error:

$$MSE = \frac{1}{n}\Sigma_{i=1}^{n}(Y_i - \hat{Y_i}^2) \tag{2}$$

The reason why I used this as a loss function instead of the categorical cross-entropy is because when I used the categorical cross-entropy loss function to predict the delay, the model rarely got it exactly right (as the delay was sometimes exactly in the middle of two different categories).

The model starts with a branch, passing the input data to two different convolutional input layers, one for delay and one for frequency. This convolutional layer performs the correlation operation on the input, allowing the model to undergo feature extraction on the signal and learn aspects of it. There is then a flatten layer to reshape the convolved outputs, followed by a fully connected hidden layer. These hidden layers both use a hyperbolic tangent activation function as it seems to work well. I chose this activation function over ReLu because I was running into the "dead ReLu" problem when trying to learn the delay. That is, my delay was always being predicted to be 0 every-time. This likely happened because the real delay was negative, and the ReLu treats negative values as being zero. Gradient descent then cannot update the weights, as the gradient of the zero function is also zero. Switching from ReLu to hyperbolic tangent solved this problem for me. Once the signal was passed through the hidden layer, it is finally put through an output layer, which reshapes it into a single number, which is the guessed output for both the frequency and the delay. A crucial thing to note is that the exact numbers of nodes in each of the layers is somewhat arbitrary, and I came to the numbers that I did through simply tinkering with these "hyper-parameters" and seeing what worked.

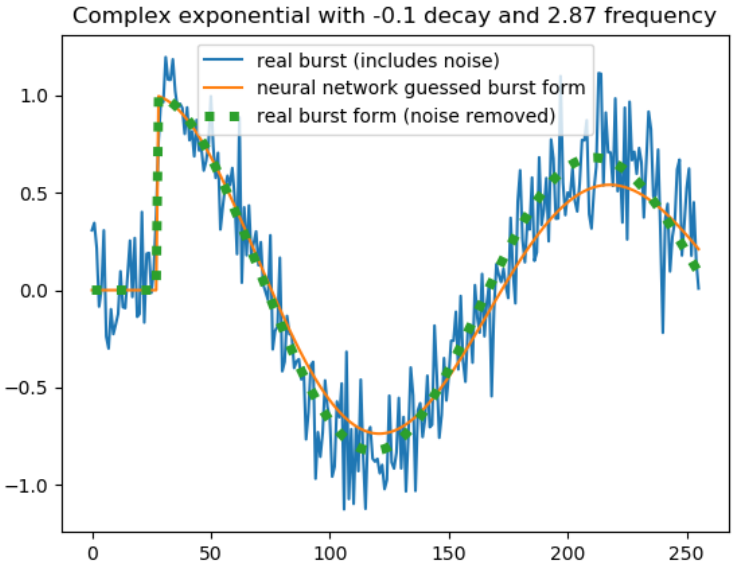
2.2 Model performance

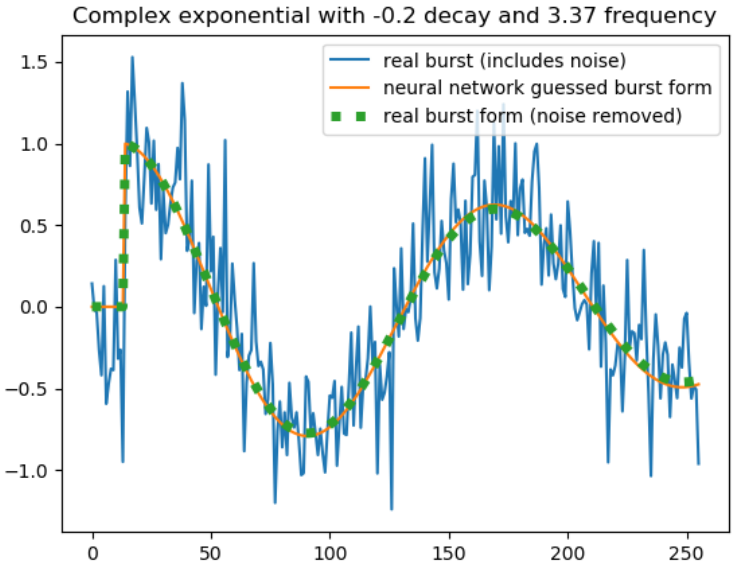
To test the model, I ran it, with the following settings. I set it to run for 7 epochs, each with a batch size of 64, and 200 training iterations

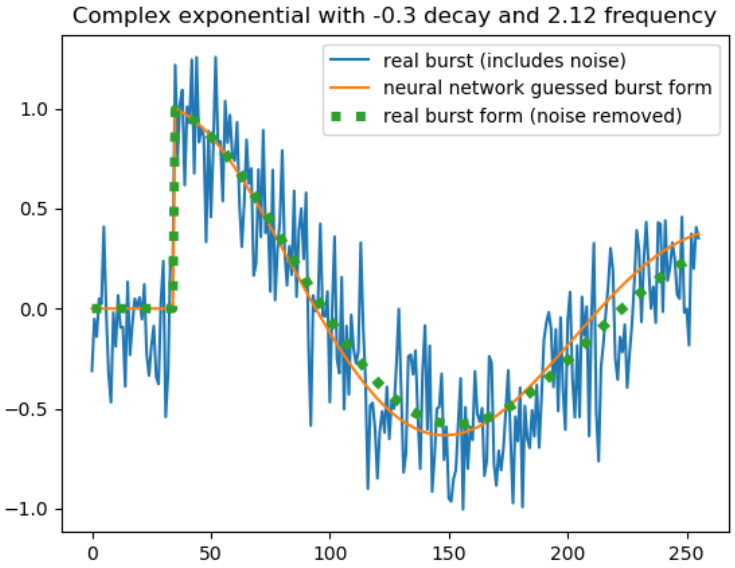
```
Model: "radio detection network"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_1 (InputLayer)         [(None, 256, 1)]          0
_________________________________________________________________________________________________
conv1d (Conv1D)              (None, 86, 12)            780         input_1[0][0]
_________________________________________________________________________________________________
conv1d_1 (Conv1D)            (None, 86, 12)            780         input_1[0][0]
_________________________________________________________________________________________________
flatten (Flatten)            (None, 1032)              0           conv1d[0][0]
_________________________________________________________________________________________________
flatten_1 (Flatten)          (None, 1032)              0           conv1d_1[0][0]
_________________________________________________________________________________________________
frequency_layer (Dense)      (None, 50)                51650       flatten[0][0]
_________________________________________________________________________________________________
decay_layer (Dense)          (None, 50)                51650       flatten_1[0][0]
_________________________________________________________________________________________________
frequency_output (Dense)     (None, 1)                 51          frequency_layer[0][0]
_________________________________________________________________________________________________
decay_output (Dense)         (None, 1)                 51          decay_layer[0][0]
=================================================================================================
Total params: 104,962
Trainable params: 104,962
Non-trainable params: 0
_________________________________________________________________________________________________
```

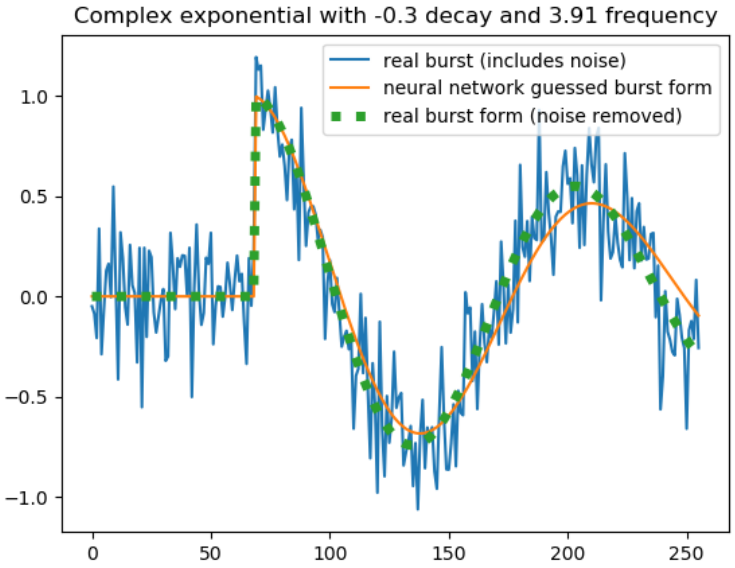
Figure 3: Neural Network Architecture

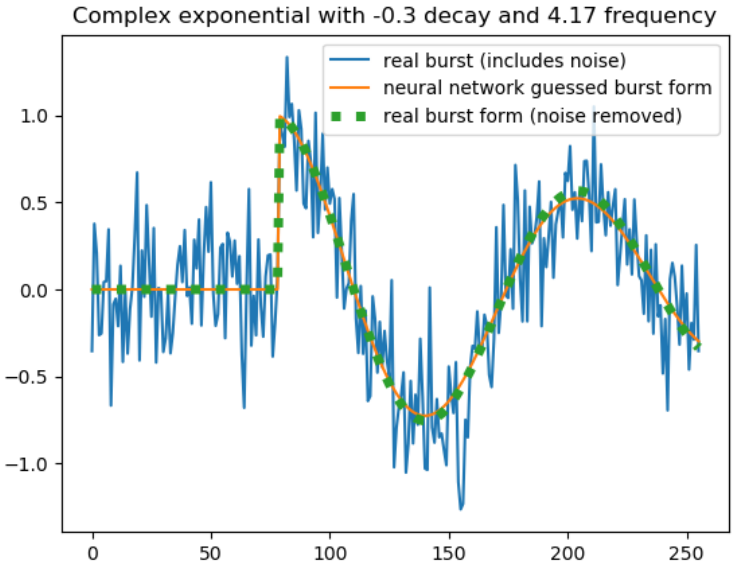
To test how well it performed after training, I fed it noisy test signals, and got it to predict the properties of the radio bursts embedded in them, plotting the results next to the real thing. As we see, the model does very well at predicting the properties of these signals, and its implementation is thus a success.

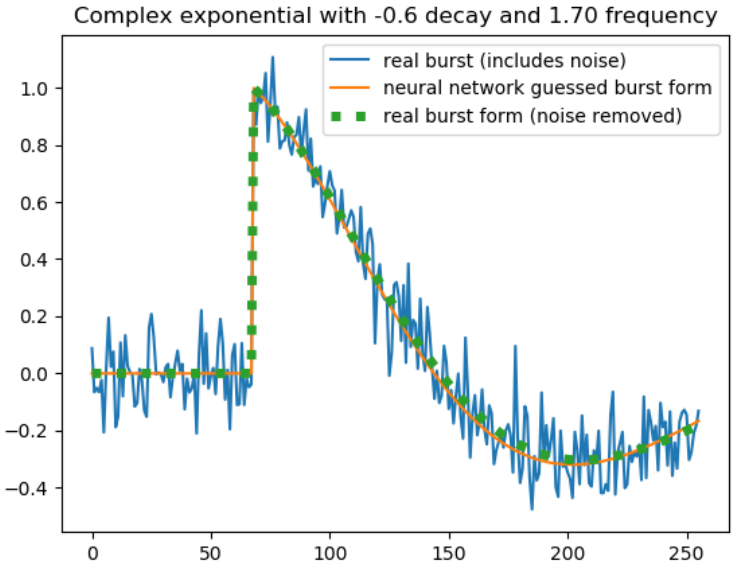

Complex exponential with -0.1 decay and 2.87 frequency
real burst (includes noise)
neural network guessed burst form
real burst form (noise removed)
1.0
0.5
0.0
−0.5
−1.0
0
50
100
150
200
250

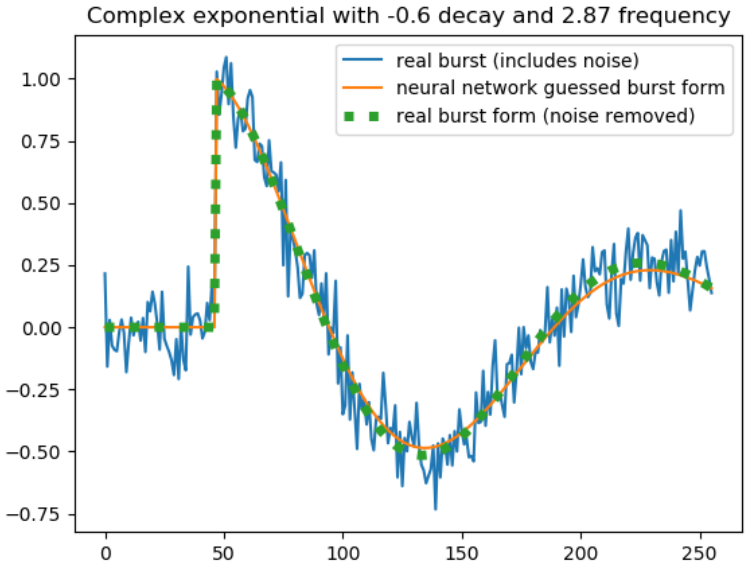

Complex exponential with -0.2 decay and 3.37 frequency
real burst (includes noise)
neural network guessed burst form
real burst form (noise removed)
1.5
1.0
0.5
0.0
−0.5
−1.0
0
50
100
150
200
250

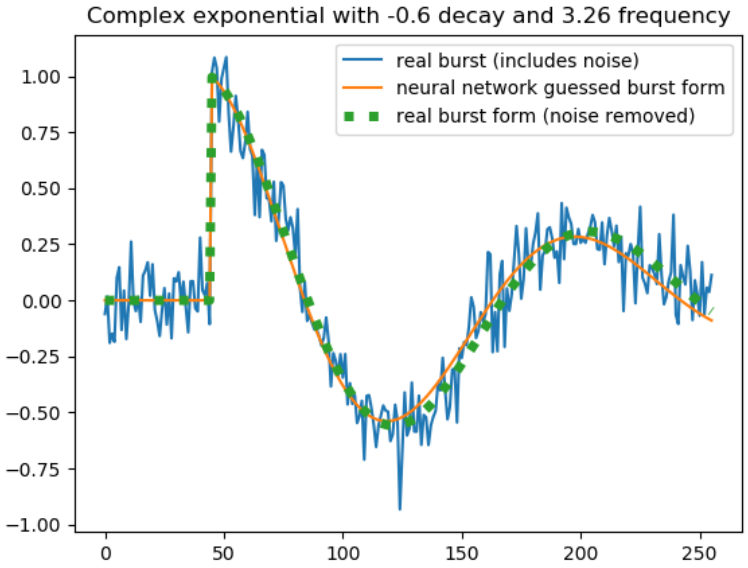

Complex exponential with -0.3 decay and 2.12 frequency
real burst (includes noise)
neural network guessed burst form
real burst form (noise removed)
1.0
0.5
0.0
−0.5
−1.0
0
50
100
150
200
250

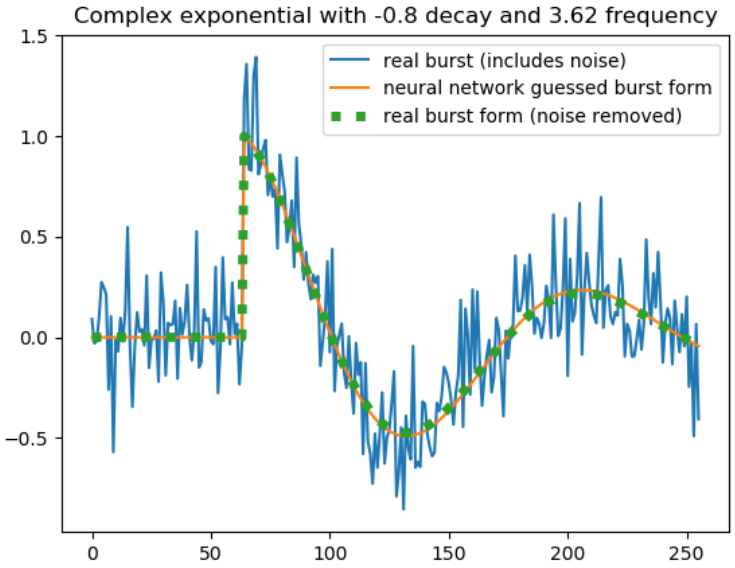

Complex exponential with -0.3 decay and 3.91 frequency
real burst (includes noise)
neural network guessed burst form
real burst form (noise removed)
1.0
0.5
0.0
−0.5
−1.0
0
50
100
150
200
250


Complex exponential with -0.3 decay and 4.17 frequency
real burst (includes noise)
neural network guessed burst form
real burst form (noise removed)
1.0
0.5
0.0
−0.5
−1.0
0
50
100
150
200
250


Complex exponential with -0.6 decay and 1.70 frequency
real burst (includes noise)
neural network guessed burst form
real burst form (noise removed)
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
0
50
100
150
200
250


Complex exponential with -0.6 decay and 2.87 frequency
real burst (includes noise)
neural network guessed burst form
real burst form (noise removed)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
50
100
150
200
250


Complex exponential with -0.6 decay and 3.26 frequency
real burst (includes noise)
neural network guessed burst form
real burst form (noise removed)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
50
100
150
200
250



References

Categorical Cross-entropy. URL: https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/categorical-crossentropy.