G2Net

Gravitational Wave Detection

Find gravitational wave signals

from binary black hole collisions

Kaggle competition

Table of Content

[Introduction 3](#_Toc77151060)

[Story 3](#_Toc77151061)

[Objective 3](#_Toc77151062)

[Inventory 4](#_Toc77151063)

[Workplan 4](#_Toc77151064)

[Notebooks 4](#_Toc77151065)

[Starter from community 5](#_Toc77151066)

[Dataset starter 5](#_Toc77151067)

[Inference 5](#_Toc77151068)

[Data exploration 5](#_Toc77151069)

[Simple EDA 6](#_Toc77151070)

[Spectrogram 7](#_Toc77151071)

[QCT 8](#_Toc77151072)

[Improvement 9](#_Toc77151073)

[Tensorflow Dataset 9](#_Toc77151074)

[Ingestion to Dataset 10](#_Toc77151075)

[Data transformation 11](#_Toc77151076)

[Back to reality 14](#_Toc77151077)

[Modeling 14](#_Toc77151078)

[Infrastructure 14](#_Toc77151079)

[Submission 16](#_Toc77151080)

[Conclusions 16](#_Toc77151081)

# Introduction

## Story

Inspired from:

<https://en.wikipedia.org/wiki/First_observation_of_gravitational_waves>

In 2015 September the 14th, the first direct observation of gravitational waves happened and officially confirmed the 11th of February 2016.

This event had been recorded by the two Laser Interferometer Gravitational-Wave Observatory (LIGO) located in the United States of America (Hanford and Livingston).

A third Large Interferometer named Virgo located in Europe was upgraded shortly after improving the Gravitational Waves potentialities.

Since then, these 3 Larges Interferometers are actively searching for detectable Gravitational Waves (GW).

In short, the interferometers are built around an L structure where laser emits photon along the few kilometer’s arms, these photons are reflected at the extremities, then is measured their travel time.

When a strong enough event happens the GW impacts photons travel time (rather curve the space structure which alters the travel length, hence modify the travel time).

By strong event here we are talking of black hole collision or at least neural star collision. Yes, this is a scary universe…

## Objective

Long story short, the objective is predicting whether 3 channels recorded events match gravitational wave. A simple yes or no binary classification problem.

Quoting Kaggle:

<https://www.kaggle.com/c/g2net-gravitational-wave-detection/data>

In this competition you are provided with a training set of time series data containing simulated gravitational wave measurements from a network of 3 gravitational wave interferometers (LIGO Hanford, LIGO Livingston, and Virgo). Each time series contains either detector noise or detector noise plus a simulated gravitational wave signal. The task is to identify when a signal is present in the data (target=1).

The parameters that determine the exact form of a binary black hole waveform are the masses, sky location, distance, black hole spins, binary orientation angle, gravitational wave polarization, time of arrival, and phase at coalescence (merger). These parameters (15 in total) have been randomized according to astrophysically motivated prior distributions and used to generate the simulated signals present in the data, but are not provided as part of the competition data.

## Inventory

### Kaggle

Kaggle provide with cloud (remote) Jupyter notebooks, 16GB RAM, multi-processor, 20GB local space.

Kaggle notebooks can be GPU enhanced but limited to 35 hours per week, counter is running for the notebooks not the GPU usage time; therefore, to be activated cautiously, that’s why the below local environment will be leveraged for most of the trainings.

Also note when running in GPU mode, less CPU are available so care with heavy CPU preprocessing, in fact we’ll rely on intermediate preprocessed data.

### Local

A local computer based on CPU Intel 4790K, 16GB RAM, GPU RTX2060, and SSD.

### Framework

Jupyter python, with TensorFlow Framework chosen as per my competencies.

Leveraging at most the TensorFlow API including Datasets, addons, and integrated Keras is a personal bet I hope will be rewarding in the future, but may be tricky at some point due to the Kaggle still in 2.4.0 Tensorflow update level.

### Data

There are 560 000 numpy records available for training.

Each data sample (npy file) contains 3 time series (1 for each detector), each spanning for 2 sec, and sampled at 2,048 Hz. Something like 96KB data per sample.

That’s around 60GB to deal with when including test data.

### Competition

Root of the competition is at <https://www.kaggle.com/c/g2net-gravitational-wave-detection>

Started the 30th of June 2021

Runs for 3 months

# Workplan

First step is identifying a starter notebook will help getting hands on Kaggle environment and uses.

the second step which is the first focus of this study is rewriting the data ingestion and transformation mechanisms matching at best Tensorflow build in Dataset mechanism.

Last in line the modeling will leverage previously generated data Tensorflow Dataset way trying to predict whether each record matches a Gravitational Wave as expected by the competition.

## Notebooks

Based on an existing starter notebook the plan is building two notebooks, the first focusing on data ingestion and transformation, the second handling the machine learning (ML) model based on the transformation schema defined in the first.

## Starter from community

« TF G2Net EDA and Starter » <https://www.kaggle.com/mrigendraagrawal/tf-g2net-eda-and-starter>

The mentioned provides with a functional notebook for data ingestion, transformation, and basic model run leveraging transfer learning from an EfficientNet B0 model.

What bothered me at first was the Dataset mechanism relying on “Sequence” imported from “keras.utils” which as per my understanding has now most of its complexity integrated now days Tensorflow Dataset API.

In line with the “Sequence”, the notebooks mix Tensorflow and former Keras functions. This may be source of confusion and as per my short experience bug friendly. Hence my will to rewrite this portion leveraging at best Tensorflow API.

Yet don’t get confused with my concerns regarding this initial starter notebook, it is a good job fully functional and definitely a good starting point permitting training and submission out of the box.

## Dataset starter

The first notebook proposed will propose rewritten data ingestion leveraging Tensorflow Dataset API.

Kaggle notebook available at <https://www.kaggle.com/vincentdumetz/g2net-tf-dataset-starter>

## Data exploration

Data exploration may happen at any stages.

There were already several public Kaggle notebooks available for this competition which inspired me.

Exploration will be located either in the Dataset Starter notebook either in a dedicated notebook for the sake of clarity and automated dataset generation unexpected alteration prevention.

## Inference

The second notebook will handle inference based on the Dataset workflow defined in previous notebook.

Kaggle notebook available at <https://www.kaggle.com/vincentdumetz/g2net-tf-dataset-inference>

# Data exploration

Let’s have a look at the data and their environment.

Below some considerations on the physical limits regarding the data at disposal.

### Limitation

<https://www.tat.physik.uni-tuebingen.de/~kokkotas/Teaching/NS.BH.GW_files/GW_Physics.pdf>

For a gravitational wave with amplitude h ∼ 10−21 and detector arm- length 4 km (such as LIGO), this will induce a change in the arm-length of about ∆L ∼ 10−16.

#### Photon shot noise.

This limitation in the detector’s sensitivity due to the photon counting uncertainty is known as photon shot noise. For a typical laser interferometer, the **photon shot noise is the dominant source of noise for frequencies above 200 Hz**, while its power spectral density Sn(f) for frequencies 100-200Hz is of the order of ∼ 3 × 10−23 √Hz.

#### Radiation pressure noise

As we have seen, the photon shot noise decreases as the laser power increases, while the inverse is true for the noise due to radiation pressure fluctuations. If we try to minimize these two types of noise with respect to the laser power, we get a minimum detectable strain for the optimal power via the very simple relation (3.10) which for the LIGO detector (where the mass of the mirrors is ∼100 kg and the arm length is 4 km), **for observation time of 1ms, gives hmin ≈ 10−23**.

#### Quantum limit

An additional source of uncertainty in the measurements is set by Heisenberg’s principle, which says that the knowledge of the position and the momentum of a body is restricted from the relation ∆x·∆p ≥ h.

Surprisingly, this is identical to the optimal limit that we calculated earlier for the other two types of noise.

#### Seismic noise

At frequencies below 60 Hz, the noise in the interferometers is dominated by seismic noise. This effect is strongly suppressed by properly designed suspension systems. **Still, seismic noise is very difficult to eliminate at frequencies below 5-10 Hz.**

#### Residual gas-phase noise

The statistical fluctuations of the residual gas density induce a fluctuation of the refraction index and consequently of the monitored phase shift. For this reason, the laser beams are enclosed in pipes over their entire length. Inside the pipes a high vacuum of the order of 10−9 Torr guarantees elimination of this type of noise.

### Conclusion

From the upper consideration we should focus on wave length from 15 to 200.

## Simple EDA

Remember records run over 2 seconds sampled at 2048Hz.

### Data range

Below are some stats worth mentioning over thousands of records

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Site | Mean Std | Min | max | Range (max-min) |
| Site 0 (LIGO 1) | 7.269105e-21 | -2.883983e-20 | 3.534220e-20 | 6.418203e-20 |
| Site 1 (LIGO 2) | 7.315387e-21 | -2.918349e-20 | 2.801213e-20 | 5.719562e-20 |
| Site 2 (Virgo) | 1.805447e-21 | -8.079410e-21 | 7.811783e-21 | 1.589119e-20 |

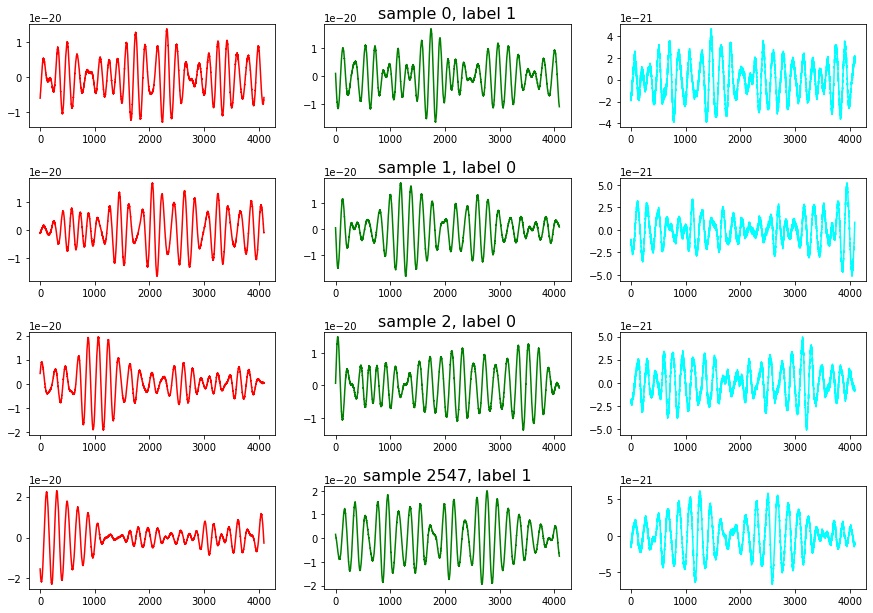
Stats are consistent between the two LIGOs but we see Virgo has different range with less amplitude and standard deviation.

Before processing data will be divided by their maximum scaling to 1 as max per channel in each recorded event.

Yet we may wonder about the impact of the Virgo being an ‘other type’ of Large Interferometer. This may deserve a special attention, yet will not be considered here further.

### Wave plot

Below are a few samples plotted, label 1 means a GW is present while label 0 means no GW.



## Spectrogram

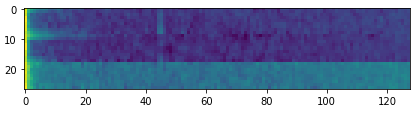
### Quote from Kaggle:

“The integrated signal-to noise ratio (SNR) is classically the most informative measure of how detectable a signal is and a typical level of detectability is when this integrated SNR exceeds ~8. This shouldn't confused with the instantaneous SNR - the factor by which the signal rises above the noise - and in nearly all cases the (unlike the first gravitational wave detection GW150914) these signals are not visible by eye in the time series.”

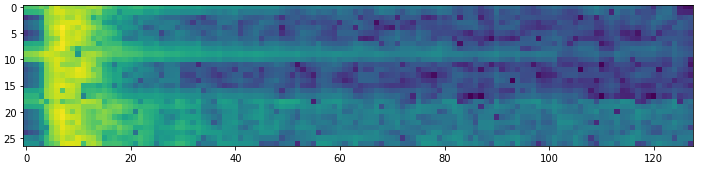
The first approach proposed in the starter was leveraging the “librosa” library transforming the data into “melspec” spectrograms.

First code did leverage the “melspectrogram” function from “librosa” I did alter a bit on “fmin” an “fmax” conforming to previous observation.

Below a example of spectrogram



With fmin and fmax adapted:



melspecs = []

for j in range(3):

melspec = librosa.feature.melspectrogram(waves[j] / max(waves[j]),

sr=4096, n\_mels=128, fmin=10, fmax=200)

melspec = librosa.power\_to\_db(melspec)

melspec = melspec.transpose((1, 0))

melspecs.append(melspec)

image = np.vstack(melspecs)

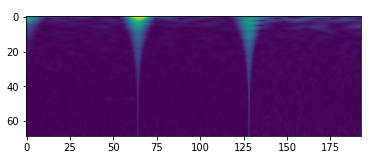
return image

By the way one should pay attention to the 3 waves being independently transformed then vertically stacked generating the final image. The result is grey scaled and coloring mapped to intensity.

Yet the Transfer Learning did perform well leveraging an Efficientnet B0 achieving over .80 Area Under the Curve which is the binary metric chosen by Kaggle for this competition.

## QCT

But few days after the owner of the starter updated with an improved signal transformation, the Constant-Q Transform (CQT) here integrated via the “nn.audio” Torch library in the “CQT1992v2” specific implementation.

The graphical output is horizontally stacked this time and the visible pikes match the jointures.:

Initial code below:

def increase\_dimension(idx,is\_train,transform=CQT1992v2(sr=2048, fmin=20, fmax=1024, hop\_length=64)): # in order to use efficientnet we need 3 dimension images

waves = np.load(id2path(idx,is\_train))

waves = np.hstack(waves)

waves = waves / np.max(waves)

waves = torch.from\_numpy(waves).float()

image = transform(waves)

image = np.array(image)

image = np.transpose(image,(1,2,0))

return image

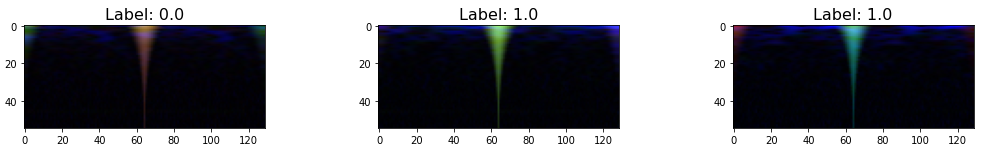
## Improvement

Until now the transformations remain grey scaled and the starter’s owner relies on a convolutional 2D layer trick in order to feed the 3 RGB channels expected by the EfficientNet for modeling.

My little idea is rather than stacking the transformations let’s populate each of the RBG channels to respective (LIGO1, LIGO2, Virgo) instruments.

Yet the visible graphical part which seems of interest as seen before with the jointure is on the extremities of the graphical output, therefore and until better mastering of these transformation techniques, I concatenate each wave with itself achieving the centering of the visual.

The output for 3 different samples becomes:



Each color matches an instrument: Red for LIGO1, Green for LIGO2, and Blue for Virgo. Yet don’t be fooled by the coloring here: They are far from conclusive detecting a Gravitational Wave.

At this stage I definitely need to improve my skills in signal analysis and transformation, but for now focusing on the modeling fixing technical and physical limitation, I’m facing:

* The transformations are CPU killer,
* Relying on intermediate pre-processed dataset I can @home but the 20GB output Kaggle limitation can’t handle the initial 60GB data transformation
* Therefore, I can hardly leverage the GPU as Kaggle diminish to 2 CPU cores when activating GPU… To be continued.
* Last but not least the CQT1992v2 is from Torch library which impose relying on external py\_function while inside the Tensorflow pipeline.

# Tensorflow Dataset

Let’s look at the Tensorflow Dataset implementation now

The main challenges are:

* Tensorflow version
  + Kaggle Tensorflow is version 2.4.1 which is quite old now and Dataset handling has been greatly improved in 2.5 (not to mention nightly tf is in 2.7)
  + Activating the GPU after upgrading Tensorflow to 2.5 requires specific CUDA libraries handling not to mention the start overhead due to tf upgrade not mentioning the packages conflict with Torch version. Long story short: I don’t have time for this road right now.
* Tensorflow Dataset do not handle the data former numpy format naturally
  + This I fixed!

That’s it let’s start with data Ingestion will help understanding the Kaggle Notebook at <https://www.kaggle.com/vincentdumetz/g2net-tf-dataset-starter>

## Ingestion to Dataset

### Sample access

Samples are delivered in the form of 560 000 distinct files of 98 432 bytes each.

There is a “training\_labels.csv” 2 columns files, first one being the id as string, second the label (0 or 1)

They are arranged in a 3 levels hierarchy based on the file name first chars, hence the below file path access function for train files:

file = return os.path.join(

BASE\_PATH, 'train',

file\_name[0],

file\_name[1],

file\_name[2],

file\_name + ".npy")

First operation creates a Pandas DataFrame enhancing the base with files path in a new ‘file\_path’ column.

### Split and shuffling

Next operation consists of splitting the previous dataset between train and validation subsets all shuffled via the “train\_test\_split” Scikit Learn (sklearn) library.

### Numpy parsing

As stated, the Numpy format is not yet handled but the below function gives us the header size:

def npy\_header\_offset(npy\_path):

with open(str(npy\_path), 'rb') as f:

if f.read(6) != b'\x93NUMPY':

raise ValueError('Invalid NPY file.')

version\_major, version\_minor = f.read(2)

if version\_major == 1:

header\_len\_size = 2

elif version\_major == 2:

header\_len\_size = 4

else:

raise ValueError('Unknown NPY file version {}.{}.'.format(version\_major, version\_minor))

header\_len = sum(b << (8 \* i) for i, b in enumerate(f.read(header\_len\_size)))

header = f.read(header\_len)

if not header.endswith(b'\n'):

raise ValueError('Invalid NPY file.')

return f.tell()

Thanks to “ <https://stackoverflow.com/questions/48889482/feeding-npy-numpy-files-into-tensorflow-data-pipeline>” by the way.

By this mean we now have constant Numpy header of size 128 and remaining data length 98 304.

The math is good:

98304 + 128 = 98432

98432 / 3 / 4096 = 8 => type is float64

### Dataset reader

In order to read unhandled raw format TensorFlow provides with the “tf.data.FixedLengthRecordDataset” function which requires the data format will be dealt by the “tf.io.decode\_raw” function, then reshaped according to expected data formatting:

ds\_train\_data = tf.data.FixedLengthRecordDataset(

df\_train['file\_path'],

98304,

header\_bytes=128,

num\_parallel\_reads=4)

ds\_train\_data = ds\_train\_data.map(lambda s: tf.reshape(tf.io.decode\_raw(s, tf.float64), (3,4096)))

Finally, we recast data to float32 type freeing some space and complying with next operations inline.

ds\_train\_data = ds\_train\_data.map(lambda s: tf.cast(s, tf.float32))

## Data transformation

### Resumé

We now have at disposal a TensorFlow Dataset once zipped with labels with the below cardinality:

<bound method DatasetV2.cardinality of <MapDataset shapes: (3, 4096), types: tf.float32>>

That is 3 waves of 4096 float32 values.

The Notebooks provides with some checks confirming data integrity via wave plotting.

### Full TensorFlow trivial

#### Tensorflow spectrogram

The below was my first attempt generating a 3D tensor dataset would match the (sizeA, sizeB, 3) RGB input format expected by most available CNN for Transfer Learning (TL).

The transformation was not meant to be efficient but shows the mechanics could be leveraged by exclusive TensorFlow pipeline.

The function “spectrogram” from the “tfio.audio” library is applied via a map on each wave:

ds\_train\_data = ds\_train\_data.map(

lambda s: tfio.audio.spectrogram(

s,

nfft=128,

window=256,

stride=64))

The input (4096) becomes a (64,65) grey scale image, with 3 waves this provides with new cardinality:

<bound method DatasetV2.cardinality of <MapDataset shapes: (3, 64, 65), types: tf.float32>>

As a matter of exercise, a log transformation is applied with clipping preventing from log(0) NaN curse, could also have been a log(x+1) all values being positive.

ds\_train\_data = ds\_train\_data.map(

lambda s: (

tf.math.log(

tf.clip\_by\_value(s,1e-30, 1e-17)) + 60)/25)

print(ds\_train\_data.cardinality)

#### To RGB

The shape is (3, 64, 65) the trick is trivial achieving an RBG like image. We perform a simple transpose of the tensor:

ds\_train\_rgb = ds\_train\_data.map(lambda s: tf.transpose(s))

Final cardinality (data only):

<bound method DatasetV2.cardinality of <MapDataset shapes: (65, 64, 3), types: tf.float32>>

This dataset can feed any known CNN.

And voila!

Each one of the 3 RGB channels consists of one Large Interferometer’s wave spectrogram transformation.

### CQT Transformation

As stated in the data exploration the starter’s owner came a few days after with a new version leveraging a CQT transformation relying on Torch libraries.

This will break the full TensorFlow paradigm but until I get better at signal transforming, I consider this a major upgrade worth implementing.

Therefore, below is described the inclusion of external functions.

#### Abstract

A few words on TensorFlow pipelines, these are not instantiated before the Dataset is fed with data. Instead, it’s a computational graph that is generated which is mainly responsible of enforcing data consistency (shapes) and preparing for GPU computation outsourcing.

This means that once in a Dataset workflow there are only tensors abstracts and their computations, which also means that inside one can only rely on TensforFlow specific functions for any calculation, transformation, etc…

What if we need external python function nevertheless?

It is possible thanks to the “tf.py\_function” but this will be at the cost of optimization.

#### Starting point

Back the numpy ingestion level from:

ds\_train\_data = tf.data.FixedLengthRecordDataset(

df\_train['file\_path'],

98304,

header\_bytes=128,

num\_parallel\_reads=4).map(wave\_transform)

#### Data preparation

The upper suggest we leverage a “wave\_transform” function will handle the mapping, so far so good:

* Decode, reshape, cast to float32 => shape (3, 4096)
* Next we split per wave (data0, data1, data2) ; note I’d like to automate this the full tensor way.
* divide each wave by it’s maximum value which range them from 0 to 1 (because all positive).
* Restack in (3, 4096) shape; see why the tensor way? I’ll definitely work on this.

#### Python function call

Now we can call the “tf.py\_function” will allow for TensorFlow’s aliens.

* + “cqt\_2\_rgb” is the function to be called.
  + [data] is the data themselves provided in list format (imagination friendly)
  + [tf.float32] is the mandatory specs of the data will be returned; remember computational graph, shape and type consistency? Here we are.

def wave\_transform(data):

data = tf.reshape(tf.io.decode\_raw(data, tf.float64), (3,4096))

data = tf.cast(data, tf.float32)

data0 = data[0]/data[0][tf.argmax(data[0])]

data1 = data[1]/data[1][tf.argmax(data[1])]

data2 = data[2]/data[2][tf.argmax(data[2])]

data = tf.stack([data0, data1, data2])

data = tf.py\_function(

cqt\_2\_rgb,

[data],

[tf.float32])

data = tf.convert\_to\_tensor(data[0])

#### CQT1992v2 transformation

Here we operate with the below function from Torch library:

import torch

from nnAudio.Spectrogram import CQT1992v2

transform=CQT1992v2(sr=2048, fmin=22, fmax=512, hop\_length=64)

Note the transformation is instanciated in the global space preventing from instancing overhead.

Worth noting here also are the specs of the transform with min and max freq, min raised to 22 because Virgo appears noisy (RGB, Blue remnants on output) when lowered. Max freq should be less but this dictates the output shape would run below the 32 minimum image size (EfficientNet specs). Anyway, seems margin for improvement here and my bet we can find more relevant transformations (story of my next life).

Then the function itself with input (3, 4096):

* Each wave concatenated with itself, which is a bit ugly but permits visible part of interest being centered (again to be worked out).
* Each wave transformed by the instanciated CQT1992v2
* Data are converted back to tensor becoming (3, 55, 129)
* Finally, the RGB trick returning an image like tensor of size (55, 129, 3)

Below the function:

def cqt\_2\_rgb(data): # in order to use efficientnet we need 3 dimension images

data = data.numpy()

wave0 = np.concatenate((data[0], data[0]))

wave1 = np.concatenate((data[1], data[1]))

wave2 = np.concatenate((data[2], data[2]))

wave0 = transform(torch.from\_numpy(wave0))

wave1 = transform(torch.from\_numpy(wave1))

wave2 = transform(torch.from\_numpy(wave2))

image = tf.convert\_to\_tensor([np.array(wave0)[0], np.array(wave1)[0], np.array(wave2)[0]])

image = tf.transpose(image,(1,2,0))

return image

Worth noting returned values range from 0 to 3 (tested over a thousands) but seems values >= 1 are rare so I rely on a ceiling at .99 matching a [0,1] range consistent with CNN inputs

Again, margins for improvements almost everywhere.

We can feed our CNNs from our own Datasets.

# Back to reality

## Modeling

I will not extend much here on the modeling, working on EfficientNet B0 to B5 for Transfer Learning with quite simple architecture.

Worth mentioning:

* BatchNormalisation on input before the EfficientNet definitely improves performances with it’s learned scaling.
* Last pooling set to ‘max’, also noticeable improvement over ‘avg’
* Top level with dropout, and 2 dense layers 64 then 32 neurons for now, with LeakyRelu activation.
* Last dense layer specialized for binary classification with one unique neuron and sigmoid activation which ranges from 0 to 1 saturating at extremities; less than 0.5 means no GW, while more than .5 means GW present in sample.

Loss function set to binary cross entropy.

Performance measurement according to Kaggle expectations with Area Under the Curve (AUC) metric.

The learning rate is handled by the exponential Decay Scheduler “tf.keras.optimizers.schedules.ExponentialDecay”, with mostly:

* Initial rate 1e-3,
* Decay 0.9,
* Steps 10 000.

## Infrastructure

### constraints

Let’s dive a bit here.

Because Infrastructure constraints are a major concern even considering I’m kinda lucky having a 4790K, 16GB, RTX2060 GPU at home.

There are 560K records for 60GB data in inputs. Reading the entire set in itself is already takes a while.

Now add up the transformation?

In first attempts with initial transformation and light modeling, full TensorFlow pipeline was achieving 5-15% SSD usage, 20-30% CPU, and 90%+ GPU.

The CQT is known computational heavy. In short only reading data and performing computation WITHOUT training raises my CPU to 100% constant and the GPU to 60% only using around 3% of SSD. Not even possible to train the model inline.

Kaggle Notebook is worst with better GPU mainly memory size permitting increase batch size, but definitely less CPU power. Worth mentioning GPU mode diminishes CPU core at disposal from 4 to 2.

### Counter measures

Since the beginning I identified the bottleneck handling this volumetry, yet did not really anticipated such a heavy transformation cost.

Luckily, I could evolve smoothly as I was progressing on the project thanks to robust initial architecture.

As I was new to Kaggle I replicated environments between home and remote on a regular basis leveraging split notebooks strategy.

* A starter handling
  + ingestion,
  + dataset,
  + transformation,
  + then saving transformed data (only parts between 50K and 200K records depending on focus ).
* An EDA notebook (only @home for analysis, transformation, POC, etc… )
* An inference notebook based on saved data from starter Notebook.

This worked well until CQT transformation and Kaggle submission attempt.

Long story short, the task of predict test data for submission from a loaded model itself takes around 2 hours.

As a matter of facts, I end up with the below workflow at this stage worked on 200K records training out of the 560K available:

* @Home starter notebook save a 200K dataset (Around 1 hour worktime)
* @Home inference notebook trains (A little hour for 1 epoch ; anyway seems badly overfitting after 1 epoch, yet only 200K records …)
  + Save model weights
* @Kaggle dedicated submission notebook
  + Load model saved @Home
  + Predict from test data ( still takes 3 hours for the 220K+ records ; GPU activation gain around 20-30% only so keeping available the 35H/week in case …)
  + Submit

# Submission

Shortly the achieved public score is 0.817 AUC with 200K records training.

# Conclusions

The public score in not good but not bad either.

The competition is young, started the 30th of July and I joined the 4th, I had not much resources at disposal assuming I’m also discovering Kaggle world at the same time.

I’m happy having managed to follow my initial plans which where:

* Leveraging as far as I could the TensorFlow Dataset workflow.
* Tricking the RGB mechanism with 3 waves records.
* Architecting a working and functional environment between @Home and Kaggle

I’m right now (14th of July) saving the full Dataset for full training and should update scoring accordingly once the learning rate schedule will have been adapted (might take a while tuning for the last 200K part …).

I’m now in position to step back a bit and brain storm next step as I intend to continue this competition and definitely have to improve my skills on signal handling, which domain I’ve been curious about since long without digging in. Time to fix this.

I don’t know yet whether the chosen RBG approach is a win. Honestly, I would not bet much on it yet it’s a worthy journey, permitting easy leveraging of Transfer Learning mechanism.

I also have in mind another approach. Considering the timeline Gravitational Waves generation from binary systems collisions there should be a specific behavior with frequency increasing, then shortly after the silence.   
This suggests me a strong compatibility with time series and Recurrent Neural Networks should be worth journey.