Machine learning for gravitational wave astrophysics

Amy Melina Welch[1]

[1] Physics Department, Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609, United States of America

E-mail: amwelch@wpi.edu

Abstract. Kilonovae are poorly understood due to lack of observations. Models generated with machine learning can fill in the knowledge gaps. This report is about such a machine learning model.

1. Introduction

1.1. Kilonovae

A kilonova is a collision that occurs either between two orbiting neutron stars or a neutron star and a black hole, creating supernova-like ejecta. Like all collisions between orbiting compact bodies, a kilonova generates gravitational waves. At the same time, a kilonova generates a lightcurve, making kilonovae a target for observation in the era of multimessenger astronomy.

1.1.1. GW170817 On the 17th of August 2017, during its second observing run, the LIGO-Virgo detector network observed a binary neutron star inspiral [1] while the electromagnetic (EM) lightcurve from the corresponding kilonova was also observed [2]. From the gravitational wave detection, it was possible to put constraints on the masses of the two neutron stars [1]. Knowing the masses allowed physicists to generate lightcurves according to models, notably the Dietrich-Ujevic model (DU17) which predicts a kilonova's EM signal based on neutron star mass [3]. The observed kilonova corresponded to the predicted lightcurve. Using various models of kilonovae, including the DU17 model and the Metzger 2017 (Me17) model, physicists could predict the event's ejecta mass and r-process nucleosynthesis yields [2].

1.2. Kilonova Models

1.2.1. DU17 Model The DU17 model predicts the luminosity of the kilonova given the constraints on its ejecta by the equation for time at which the region becomes visible,

$$t_c = \sqrt{\frac{\theta_{ej}\kappa M_{ej}}{2\phi_{ej}(v_{max} - v_{min})}}$$

[3], where ν_{max} ν_{min} are the maximum and minimum speeds of the ejecta, θ_{ej} and ϕ_{ej} are angles of ejecta and M_{ej} is the ejecta mass. This allows one to obtain the bolometric luminosity

$$L_{bol}(t) = (1 + \theta_{ej})\epsilon_{th}\epsilon_0 M_{ej}$$

where $\frac{t}{t_c}(\frac{t}{1d})^{-\alpha}$ for $t \leqslant t_c$ and $(\frac{t}{1d})^{-\alpha}$ for $t > t_c$ and ϵ is the specific heat for energy release due to radioactive decay. [3]. From here, one can calculate the bolometric magnitude

$$M_{bol} \approx 4.74 - 2.5log_{10}(\frac{L_{bol}}{L_{\odot}})$$

[3] The magnitude in each band of the lightcurve (denoted by X) is given by

$$M_X(t) = M_{bol}(L(t)) - BC_X(t)$$

[3], where C_X is the magnitude in a given band X, from which one can fit the parameters for the polynomials.

1.2.2. Me17 Model The Me17 Model has some similarities to the DU17 Model. However, it calculates the luminocity as

$$L_\nu = \frac{E_\nu}{t_{d,\nu} + t_{lc,\nu}}$$

where E_ν is the thermal energy. [4] It calculates the kilonova's thermal emission temperature as

$$T_{eff} = (\frac{L_{tot}}{4\pi\sigma R_{ph}^2})^{1/4}$$

where R_{ph} is the radius. The opacity

$$\kappa_\nu$$

of each layer depends on the temperature

$$T_\nu \simeq (\frac{3E_\nu}{4\pi a R_\nu^3})^{1/4}$$

[4] The model determines the kilonova's time-evolution and thermal emission properties by the equation

$$\frac{dE_\nu}{dt} = -\frac{E_\nu}{R_\nu}\frac{dR_\nu}{dt} - L_\nu + \dot{Q}$$

[4]

1.3. Machine Learning
Machine learning is a type of computer algorithm that iterates through data over a series of many epochs to better categorize the data. It allows for data categorization in ways that humans might not think to, and with data sets that are impractical to categorize by hand.

1.3.1. Normalised Flows Normalised Flows are a subset of machine learning that transform the input data into a spectrum along a Gaussian curve [5] and transform the curve to get the best fit for the data.It is necessary for machine learning that results in a lightcurve [5].

1.3.2. Machine Learning for Generating Kilonova Lightcurves Because GW170817 is the only observation of a kilonova lightcurve found in nature, the machine learning model relies on data generated through models. It uses the established parameters to generate thousands of lightcurves for training, which it uses to generate an 'ideal' lightcurve with the normalised flow model [5]. The initial algorithm in use during this project generated these lightcurves based on the DU17 Model [5], but with a few modifications to the code, it could also generate lightcurves based on the Me17 Model.

2. Motivation

2.1. Neutron Stars

Neutron stars are the dense remnants of supernovae. Their internal equation of state (EOS) is not entirely known, although given their density, it seems that they consist of a quark-gluon plasma [6].

2.1.1. Oppenheimer-Volkoff and Neutron Star EOS In 1939, the physicsists Oppenheimer and Volkoff used relativity to correct the equations of an isotropic fluid [6] in attempt to understand stellar structure [7]. Oppenheimer and Volkoff assumed spherical symmetry, resulting in the Einstein field equations

$$8\pi p = e^{-\lambda}(\frac{v'}{r} + \frac{1}{r^2}) - \frac{1}{r^2}$$

$$8\pi\rho = e^{-\lambda}(\frac{\lambda'}{r} - \frac{1}{r^2}) + \frac{1}{r^2}$$

$$\frac{dp}{dr} = -\frac{(p+\rho}{2}\nu'$$

[7] with pressure p, radius r and energy density ρ. Combining these field equations with the Schwartzschild solution yields the equation

$$\frac{dp}{dr} = -\frac{p+\rho(p)}{r(r-2u)}(4\pi pr^3 + u)$$

[7] with energy u. Today this equation is known as the Tolman-Oppenheimer-Volkoff equation (Tolmann independently derived this equation around the same time as Oppenheimer and Volkoff) or the equation of hydrostatic equilibrium. It describes neutrton stars, but in order to get a closed set of equations for describing the stars' matter, it requires the EOS, for giving the energy density in terms of the energy density and specific entropy:

$$p = p(\rho, S)$$

, although in these situations the entropy S can often be overlooked [8]. Oppenheimer and Volkoff used Fermi statistics to get the EOS

$$\rho = K(\sinh t - t)$$

$$p = \frac{1}{3}K(\sinh t - 8\sinh\frac{1}{2}t + 3t$$

where

$$K = \frac{\pi\mu_0^4 c^5}{4h^3}$$

and

$$t = 4\log\frac{p}{\mu_0 c} + [1 + (\frac{p}{\mu_0 c})^2]^{\frac{1}{2}}$$

[7]. However, this EOS gives the neutron star a maximum mass of $0.7M\odot$, which is less than the Chandrasekhar limit [6], making this EOS useless for describing real-world phenomena. However, it turned out that their EOS was limited in that it did not account for nuclear interactions, which would increase the matter's energy and therefore its mass [6], since energy and mass are equivalent by the equation

$$E = mc^2$$

. Since then, a variety of models that take nuclear interactions into account, with different equations of state, have been proposed. Each makes different predictions about internal energy.

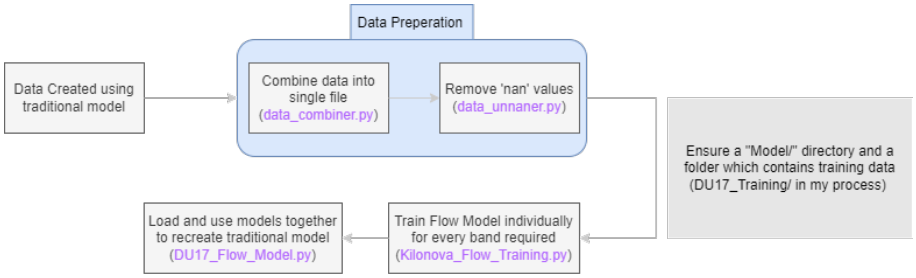

Figure 1. A flow chart of the programmes used in the machine learning process. Credit to Thomas Wallace.

2.2. Kilonovae and Nuclear Astrophysics

Elements heavier than iron, which stars cannot produce through nuclear fusion, can only be produced through two known processes, the r-process and s-process [2]. Given the abundances of various types of heavy elements [2], about half of all heavy elements must be synthesized via the r-process, or rapid neutron capture in a neutron-rich environment [4]. It is thought that binary neutron star mergers create these environments, and may produce most or all of the r-processes in the Milky Way [2]. If this is the case, however, the ejecta produced in binary neutron star collisions must give rise to environments that the r-process occurs in. The lightcurves from kilonovae can be used to determine the EOS of the ejecta.

2.3. Generating Kilonova Lightcurves

The best way to determine the EOS of the ejecta would be to observe kilonova lightcurves. Unfortunately, there have not been enough observations to make any conclusions. Thus, models are the best way to determine the EOS of the ejecta given current resources. Previously, models were slow to generate kilonova lightcurves, but machine learning allows for the rapid generation of multiple lightcurves [5].

3. Methods

This project required running code written by Thomas Wallace. The programmes had to be run in a particular order such that the computer would first generate training data with the programme $DU17_Model.py$, then prepare the data by combining it into a single file, next train on the data to study the bands that make up the lightcurve, and finally generate models (figure 1). [5]

3.1. Proof-of-Concept for DU17 Model

First, it was essential to complete a machine learning cycle by running the programmes on the machine that was to be used for this project. This did require altering some of Wallace's code given the specifications of the machine. Notably, since the programme $DU17_Models.py$ took too long to run for the purposes of a proof-of-concept run, lines 199-202 were altered to split into 25*N threads instead of 1*N threads (see Appendix A) so that the programme took less time to complete at the cost of generating a significant amount of data. Further, later programmes that generated graphics were altered to use GPU instead of cuda because the machine used for the proof-of-concept was not cuda-compatible.

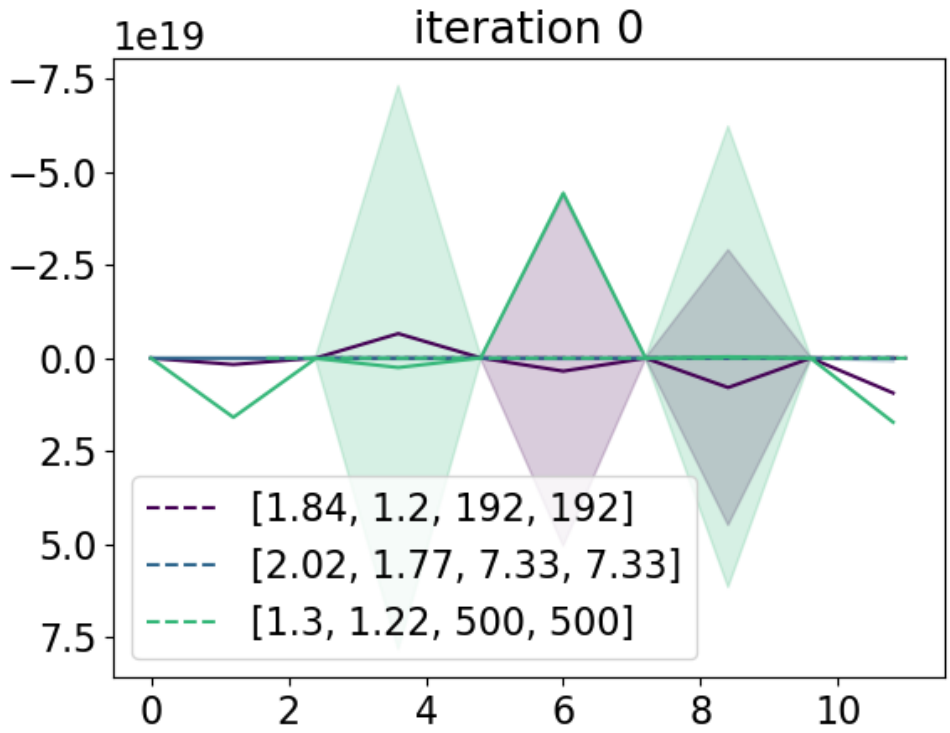

Figure 2. Band from $Kilonova_flow_training.py$ for DU17 proof-of-concept iteration 0.

3.2. Proof-of-Concept for Me17 Model

The next step was to alter the programmes to use the Me17 model instead of the DU17 model, using alternative packages from gwemlightcurves. Attepts to alter the code for this model and run the machine learning on them were successful. Unfortunately, the Me17 code was lost after running, so machine learning with this model beyond the proof-of-concept will require re-altering the DU17 code.

3.3. Further Training with DU17 Model

The next step was to train the computer with a more complete data set, achieved by splitting the data into 1*N threads where the code had previously been altered to split into 25*N threads (see Appendix A). Such that the programme could split the data into 1*N threads while still terminating in a reasonable amount of time on the machine being used, the programme was run through University of Glasgow's remote terminal cloud computing system Wiay. This generated a more complete data set, but caused problems in terms of rendering plots graphically.

4. Results

4.1. Proof-of-Concept Results

Because the DU17 model proof-of-concept generated less training data than the other iterations, the machine learning algorithm had less to work with when generating lightcurves. When the

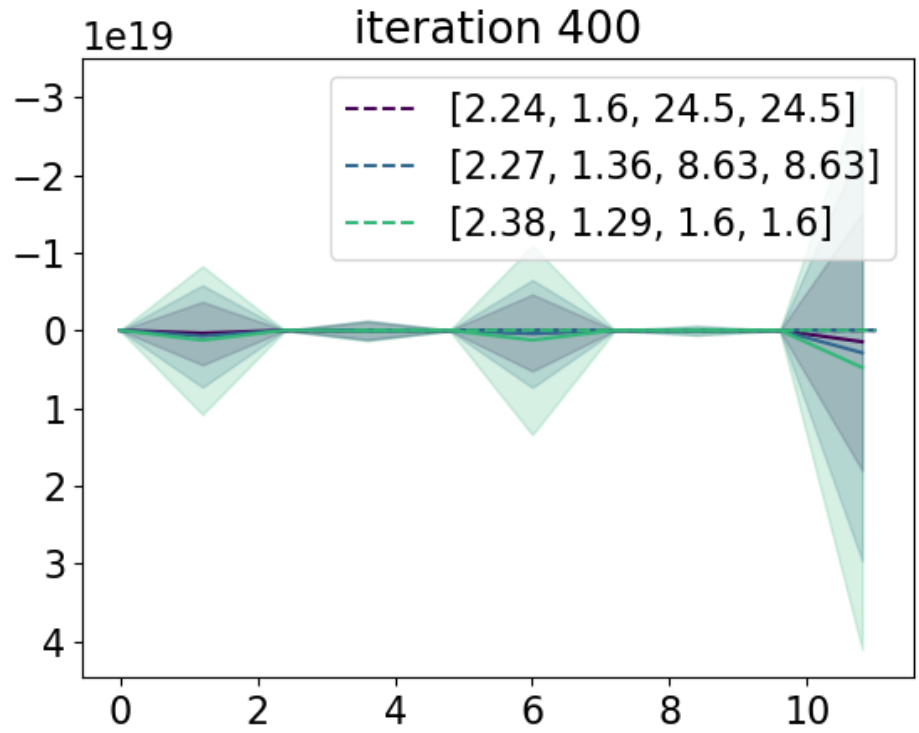

Figure 3. Band from $Kilonova_flow_training.py$ for DU17 proof-of-concept iteration 400.

machine ran the programme $Kilonova_flow_training.py$ (see Appendix C), which uses machine learning on the data produced in $DU17_Model.py$ to generate model bands of light, it generated bands that experienced little evolution over many epochs before the machine determined that it had reached the optimal band model and terminated (See figures 2-4). Thus the machine could not produce a meaningful lightcurve prediction as in figure B1 (see Appendix B). The proof-of-concept for the Me17 model also generated less data than other iterations, so it encountered the same problems as the DU17 model proof-of-concept, but the machine did not save the bands generated from this iteration.

4.2. Later Results

Later iterations of this machine learning programme were run on Wiay. Due to some issues in Wiay, results generated in later iterations of the programme for the DU17 model were never saved or even made visible to humans. However, the programmes were proven to run in Wiay.

5. Conclusions

Due to the lack of significant results, the only conclusion from this project is that the proof-of-concept data set generated by splitting into 25*N threads instead of 1*N threads (see Appendix A) does not work to generate accurate lightcurve models. More research is needed to generate better model lightcurves.

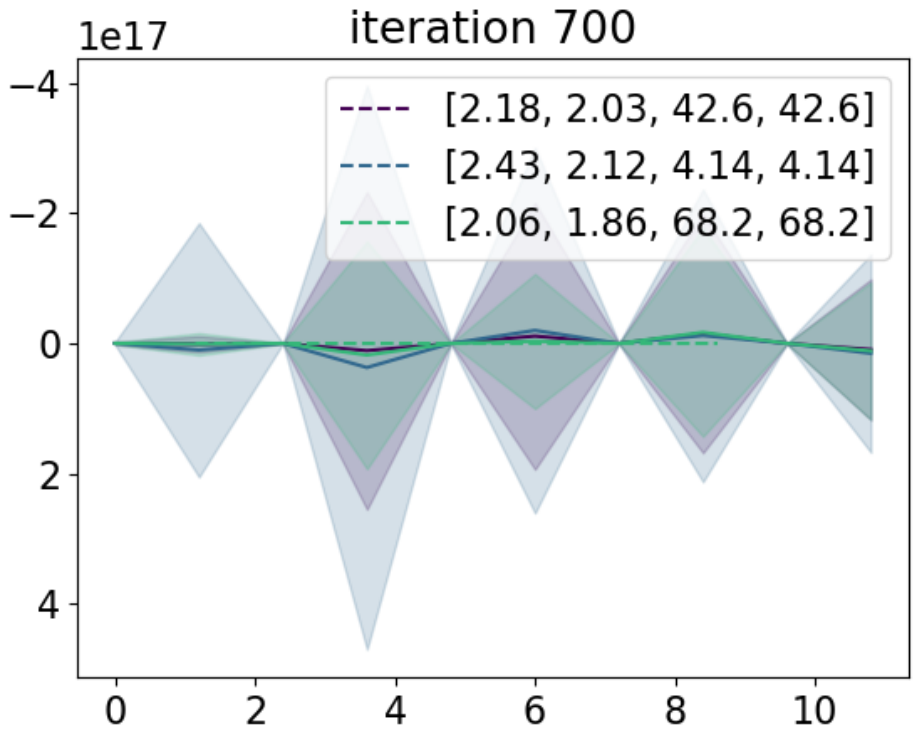

Figure 4. Band from $Kilonova_flow_training.py$ for DU17 proof-of-concept iteration 700. Note the lack of significant evolution over epochs.

6. Future Work

Possible future work on this project would entail generating complete data sets for both the DU17 and Me17 models, training the machine separately on each data set, and comparing the model lightcurves generated from each kilonova model.

Acknowledgments

Thank you to NSF grants 1460803 and 1950830 and the University of Florida for sponsoring this project. Thank you Professor Peter Wass and Professor Paul Fulda for organizing this project. Thank you Professor Ik Siong Heng for overseeing this project. Thank you Institute for Gravitational Research (IGR) graduate students Jess Irwin, Federico Stachurski and Michael Williams for their help with this project. Thank you Jamie Scott for IT support. Finally, thank you to all IGR graduate students and fellow IREU students for their comradery.

ORCID iDs

Amy Melina Welch `https://orcid.org/0000-0003-0958-3014`

Appendix A. *DU17$_M$odel.py*
Code for programme *DU17$_M$odel.py*. Credit Thomas Wallace. Note places where code was altered for various purposes, as detailed in Methods.

```
    import numpy as np
from gwemlightcurves.KNModels import table
import gwemlightcurves.EjectaFits.DiUj2017 as du
from gwemlightcurves.KNModels.io.DiUj2017 import calc_lc
import matplotlib
matplotlib.use("TkAgg")
import matplotlib.pyplot as plt
import random
import sys
from threading import Thread
import h5py
import pandas as pd

#creates lightcurve dataframes from input data using the Dietrich Ujevic 2017 model codebase
#to make with other models similair programmes to this one can be made using different parts

def Generate_LightCurve(m1,m2,l1,l2,plot = False):
    """A function to generate lightcurves based off of the two apparent masses of the neutro
            -m1,m2,l1,l2: Float inputs
            -plot: boolean for whether or not to plot the generated lightcurves
    """
    #initial parameters for the lightcurve
    tini = 0
    tmax = 11
    dt = 0.01
    kappa = 10
    eps = 1.58e10
    alp = 1.2
    eth = 0.5

    vmin = 0.02

    #calculate c1 and c2 from l1,l2
    c1 = table.CLove(l1) #Compactness-Love relation for neutron stars
    c2 = table.CLove(l2)

    #calculate M_ej
    mb1 = table.EOSfit(m1,c1)
    mb2 = table.EOSfit(m2,c2)
    mej = du.calc_meje(m1,mb1,c1,m2,mb2,c2)

    #calculate v_ej
    v_rho = du.calc_vrho(m1,c1,m2,c2)
    v_z = du.calc_vz(m1,c1,m2,c2)
    vej = du.calc_vej(m1,c1,m2,c2)

    #calculate angles
```

```
    th = du.calc_qej(m1,c1,m2,c2)
    ph = du.calc_phej(m1,c2,m2,c2)

    t_d, lbol_d,mag_new = calc_lc(tini,tmax,dt,mej,vej,vmin,th,ph,kappa,eps,alp,eth,flgbct =
    #t_d is time in days. lbol_d is bolometric luminosity, mag_new is the magnitude which we
    if plot == True:
        u = mag_new[0]
        g = mag_new[1]
        r = mag_new[2]
        i = mag_new[3]
        z = mag_new[4]
        y = mag_new[5]
        J = mag_new[6]
        H = mag_new[7]
        K = mag_new[8]

        plt.style.use("bmh")
        plt.subplot(121)
        plt.plot(t_d,u,label = "u")
        plt.plot(t_d,g,label = "g")
        plt.plot(t_d,r,label = "r")
        plt.plot(t_d,i,label = "i")
        plt.plot(t_d,z,label = "z")
        #plt.xscale("log")
        plt.xticks(np.arange(tini,tmax))
        plt.gca().invert_yaxis()
        plt.legend(prop={'size': 6})

        plt.subplot(122)
        #plt.plot(t_d,y,label = "y")
        plt.plot(t_d,J,label = "J")
        plt.plot(t_d,H,label = "H")
        plt.plot(t_d,K,label = "K")
        plt.xticks(np.arange(tini,tmax))
        plt.gca().invert_yaxis()
        plt.legend(prop={'size': 6})
        #plt.xscale("log")
        plt.show()
    return([m1,m2,l1,l1],np.array([t_d,mag_new]))#useful to return inputs



def generate_data(data):
    "function to generate multiple lightcurves based on a give dataset of inputs"
    output = []
    for i in np.arange(len(data)):
        line = data[i]
        if "idlelib" not in sys.modules:
            print(f'\r{100*i/len(data):.3f}% finished',end = '\r')
        m1,m2,l1,l2 = line
```

```
        temp_in,temp_out = Generate_LightCurve(m1,m2,l1,l2)
        output.append([temp_in,temp_out])
    return(output)



def thread_fn(m1,m2,l1,l2,fname,printing = False):
    "function used for multithreading the process. Speeds up data creation significantly"
    final_data = []
    L = len(m1)
    for i in np.arange(L):
        if printing == True:
            if "idlelib" not in sys.modules:#if running in a command or execution window
                print(f'\r{100*i/L:.3f}% finished',end = '\r')
            else: #if just running from IDLE
                if not i % 1000:
                    print(f'{i}/{L}\t{100*i/L:.2f}%')

        for x in np.arange(1):#if adding noise you would have np.arange(n) for number of poi
            #When adding noise consideer: m1 > m2 => l1 < l2
            #These comments were intiial attempts to add noise
            m1_ = m1[i] #+ random.uniform(0,0.01)*m1[i]
            m2_ = m2[i] #+ random.uniform(-0.01,0)*m2[i]
            l1_ = l1[i] #+ random.uniform(-0.01,0)*l1[i]
            l2_ = l2[i] #+ random.uniform(0,0.01)*l2[i]

            conditions,lightcurves = Generate_LightCurve(m1[i],m2[i],l1[i],l2[i])
            t_d,curves = lightcurves
            m1_,m2_,l1_,l2_ = conditions
            g = curves[1]
            r = curves[2]
            I = curves[3]
            z = curves[4]

            d = np.array([m1_,m2_,l1_,l2_,t_d,g,r,I,z])
            final_data.append(d)

    final_data = np.array(final_data)
    df = pd.DataFrame(data = final_data,
                      columns = list(['m1','m2','l1','l2','time','g','r','i','z']))

    df.to_pickle(f"{fname}.pkl")
    print(f"{fname} done")
    #returning from a thread is quite confusing so it's better to just save the individual f

def thread_fn2(fname,i,printing = False):
    "The second threading function which attempts to add noise. Was never used later in the

    print(f'thread {i} starting')
```

```
    final_data = []
    data = np.array(pd.read_pickle(fname).values)
    t = 0

    for d in data:
        L = len(data)
        t += 1
        if printing == True:
            if "idlelib" not in sys.modules:
                print(f'\r{100*t/L:.3f}% finished',end = '\r')
            else:
                if not i % 1000:
                    print(f'{t}/{L}\t{100*t/L:.2f}%')
        m1,m2,l1,l2,t_d,g,r,I,z = d
        for j in np.arange(5):
            m1_ = m1 + random.uniform(0,0.01)*m1 #m1 > m2 => l1 < l2
            m2_ = m2 + random.uniform(-0.01,0)*m2 #add a random value between m2 and m2 - 1%
            l1_ = l1 + random.uniform(-0.01,0)*l1
            l2_ = l2 + random.uniform(0,0.01)*l2
            new_d = np.array([m1_,m2_,l1_,l2_,t_d,g,r,I,z])
            final_data.append(new_d)

    final_data = np.array(final_data)

    #print(final_data[0])
    df = pd.DataFrame(data = final_data,
                      columns = list(['m1','m2','l1','l2','time','g','r','i','z']))
    df.to_pickle(f"{fname}_noise.pkl")
    print(f'thread {i} finished')


if __name__ == "__main__":#if not being imported to another python file.
    """NB: Data creation took a long time (at least 2 hours if not more if I remember correc
            it might be possible to take this code and instead of multithreading create data
            in sections."""

    #Making the first data
    filedir = "mass_lambda/mass_lambda_distributions.h5"#the file containing the input m1,m2

    d = h5py.File(filedir, 'r')
    data = np.array(d.get('labels'))
    d.close()

    m1 = data[:,0]
    m2 = data[:,1]
    l1 = np.exp(data[:,2])
    l2 = np.exp(data[:,3])

    N_threads = 16 #depends on processor being used. More threads the better. Can result in
```

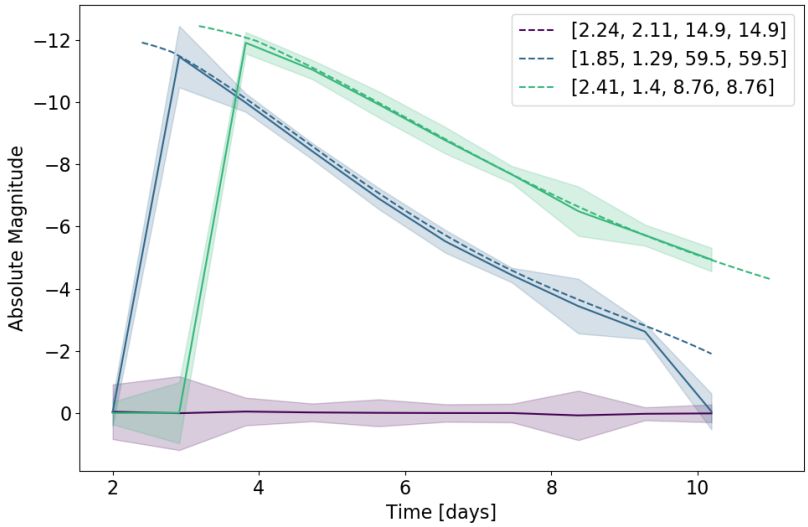

Figure B1. Example predicted lightcurve from [5]. Contrast this figure with the results of the project outlined in this report.

```
#split the inputs into constituent parts for multithreading.
part_m1 = np.split(m1, 1*N_threads) #change the multiple if you want larger splits, note
part_m2 = np.split(m2, 1*N_threads) #it can be useful to create each part seperatlety th
part_l1 = np.split(l1, 1*N_threads) # i+= n (see line 211) later for whatever segment n
part_l2 = np.split(l2, 1*N_threads) # overheating for a long period of time
threads = list()

for i in np.arange(N_threads):
    #launch all the threads
    printing = False
    if i == 0:
        #only have printing = True for the first thread to act as an indication of how l
        printing = True
    #i+=3 #if N_threads = 1 but the split is greater than that then you can add this to
    x = Thread(target = thread_fn, args = (part_m1[i],part_m2[i],part_l1[i],part_l2[i],
                                           f'DU17_training/DU17_{i}',printing,
    threads.append(x)
    x.start()

for thread in threads:
    thread.join()
print("All threads finished")
```

Appendix B. Wallace Lightcurve Model
An earlier iteration of the machine learning algorithm used generated three example lightcurve predictions (Figure B1).

Appendix C. *$Kilonova_flow_training.py$*
Code for *$Kilonova_flow_training.py$*. Credit Thomas Wallace.

```
    import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import os
import time
from matplotlib import cm
plt.style.use('seaborn-colorblind')

from glasflow import RealNVP
import torch
from torch import optim

import random
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle

from DU17_Model import Generate_LightCurve


#       HYPER PARAMS           #
#------------------------------#
troubleshooting = False #if True you can enter into trouble shooting loops

#basic hyperparameters for the machine learning
epochs = 2000
learning_rate = 1e-5
test_split = 0.33
batch_size =  1000
patience = 0.1
#for learning scheduling
step_f = 0.33
gamma = 0.01

#which band we are training to
band = 'r'
bandindex = ['g','r','i','z'].index(band) + 1

#idiot variables
axis = 0

#         SETUP 1                  #
#----------------------------------#
#                                  #
```

```
training_data = os.listdir("DU17_Training") #data is stored in this folder but typically is
loss = dict(train=[],val=[],delta=[]) #initialise loss dict

device = torch.device('cuda')#set device as GPU, change cuda to cpu if no GPU installed

#                                SCALING                         #
#----------------------------------------------------------------#
#  finding the scaling constant to use to normalise the curves  #

fname = "Data_Cache/combined.pkl"
data = pd.read_pickle(fname)
data = shuffle(data)

curve = data[band].values
curve = np.vstack(curve)
curve = np.nan_to_num(curve)

scaling_constant = np.min(curve)
print(f'scaling_constant {band}: {scaling_constant}')

#scaling_constant = -14.019296288484181 # for g band, just incase I lose it


#                   SETUP 2             #
#---------------------------------------#

fname = "DU17_Training/" + training_data[0]
print(f'file: {fname}')

data = pd.read_pickle(fname)
data = shuffle(data) #shuffling for fun

curve = data[band].values
curve = np.vstack(curve)#more convenient for manipulation

#set the Flow AI
flow = RealNVP(
    n_inputs=len(curve[0]),#based on length of training data
    n_transforms =8,
    n_conditional_inputs=4,
    n_neurons=32,
    batch_norm_between_transforms=True)

#send to GPU
flow.to(device)

#optimiser/scheduler setup
optimiser = torch.optim.Adam(flow.parameters(),lr = learning_rate)
scheduler = optim.lr_scheduler.StepLR(optimiser, step_size=step_f*epochs, gamma=gamma)
print(f'Created flow and sent to {device}')
```

```
#Ceck that the curves are correctly normalised
curve = curve/scaling_constant
try:
    assert np.max(curve) <= 1.0
except:
    print("Curve not normalised correctly, curve max was:\t",np.max(curve))

m1 = data['m1']
m2 = data['m2']
l1 = data['l1']
l2 = data['l2']
t_d = data['time']
t_d = np.vstack(t_d)[0]
conditional = np.vstack((m1,m2,l1,l2)).T
print(len(m1)," Data points")
m1 = np.vstack(data['m1'])
m2 = np.vstack(data['m2'])
l1 = np.vstack(data['l1'])
l2 = np.vstack(data['l2'])


#                  CONVERTING DATA TO TENSORS   DO NOT TOUCH               #
#-------------------------------------------------------------------------#
#           I don't fully understand what's happening here so best        #
#                         to just leave it as is, it works                #

data = []
curve_train,curve_val,conditional_train,conditional_val = train_test_split(
    curve,conditional,test_size = test_split,shuffle = False)

y_train = conditional_train
x_train = curve_train
y_val = conditional_val
x_val = curve_val

x_train_tensor = torch.from_numpy(x_train.astype(np.float32))
y_train_tensor = torch.from_numpy(y_train.astype(np.float32))
train_dataset = torch.utils.data.TensorDataset(x_train_tensor,y_train_tensor)
train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size= batch_size, shuffle = False
    )

x_val_tensor = torch.from_numpy(x_val.astype(np.float32))
y_val_tensor = torch.from_numpy(y_val.astype(np.float32))
val_dataset = torch.utils.data.TensorDataset(x_val_tensor, y_val_tensor)
val_loader = torch.utils.data.DataLoader(
    val_dataset, batch_size=batch_size, shuffle=False
)
```

```
#                  TRAINING               #
#-----------------------------------------#
print("beginning training")
for i in range(epochs):
    flow.train()
    train_loss = 0.0
    for batch in train_loader:
        x,y = batch
        x = x.to(device)
        y = y.to(device)
        optimiser.zero_grad()
        _loss = -flow.log_prob(x,conditional = y).mean()
        _loss.backward()

        optimiser.step()

        train_loss += _loss.item()
    loss['train'].append(train_loss/len(train_loader))

    flow.eval()
    val_loss = 0.0
    for batch in val_loader:
        x,y, = batch
        x = x.to(device)
        y = y.to(device)
        with torch.no_grad():
            _loss = -flow.log_prob(x,conditional=y).mean().item()

        val_loss += _loss
    loss['val'].append(val_loss / len(val_loader))

    scheduler.step()

    #Run every 10 epochs
    if not i % 10:
        try:
            print(f"Epoch {i} - train: {loss['train'][-1]:.4g}"+
                    f"\t val: {loss['val'][-1]:.4g}"+
                    f"\tlr: {scheduler.get_last_lr()[0]:.3g}"+
                    f"\tLoss: {loss['val'][-11] - loss['val'][-1]:.4g}")
            try:
                loss['delta'].append(loss['val'][-11] - loss['val'][-1])
                delta = loss['delta'][-5:]
                if all(d < patience for d in delta):
                     if all( d > -1*patience for d in delta):
                        print("Early Stopping")
                        break
            except Exception as e:
                pass
```

```
    except Exception as e:
        print(e)
        print(f"Epoch {i} - train: {loss['train'][-1]:.4g}"+
                f"\t val: {loss['val'][-1]:.4g}"+
                f"\tlr: {scheduler.get_last_lr()[0]:.3g}")

if i % 50 == 0:
    #   TESTING THE AI ON RANDOM DATA #
    #---------------------------------#
    #
    test_array = []
    indices = []
    N = 3 #number of graphs to predict
    for n in np.arange(N):
        j = random.randint(0,len(m1))
        indices.append(j)
        temp = random.choice(conditional)
        test_array.append(temp)

    test_array = np.array(test_array)

    cond = torch.from_numpy(test_array.astype(np.float32)).to(device)

    Big_Samples = []
    N_Samples = 100
    cond = torch.from_numpy(test_array.astype(np.float32)).to(device)

    #create N samples
    with torch.no_grad():
        for j in np.arange(N_Samples):
            samples  = flow.sample(N,conditional = cond)
            Big_Samples.append(samples)
    for j in np.arange(len(Big_Samples)):
        Big_Samples[j] = Big_Samples[j].cpu().numpy()



    Big_Samples = np.array(Big_Samples)

    final_samples = np.mean(Big_Samples,axis = axis)

    std = np.std(Big_Samples,axis = axis)
    max_lines = final_samples + 3*std #np.max(Big_Samples,axis = 0)
    min_lines = final_samples - 3*std #np.min(Big_Samples,axis = 0)


    cond = cond.cpu().numpy()
    #print(cond)
    cmap =cm.get_cmap('viridis') #so we can set the colour of all the plots
```

```
        for n in np.arange(N):
            #print(f"plotting {n}")
            col = cmap(n/N)
            m1_,m2_,l1_,l2_ = cond[n]
            lc = Generate_LightCurve(m1_,m2_,l1_,l2_)[1]
            #print(m1_,m2_,l1_,l2_)
            lc = np.nan_to_num(lc)
            #print(lc[1][bandindex])
            plt.plot(lc[0],lc[1][bandindex],"--",label = f"[{m1_:.3g}, {m2_:.3g}, {l1_:.3g},
            plt.plot(t_d,scaling_constant*final_samples[n],"-",ms =4,c = col)
            plt.fill_between(t_d,min_lines[n]*scaling_constant,max_lines[n]*scaling_constant

        plt.gca().invert_yaxis()
        plt.title(f'iteration {i}')
        plt.legend()
        plt.savefig(f'Model Evolution/iteration {i}.png')
        #plt.show()

        plt.clf()
print("Finished training")

#           EVALUATION          #
#-------------------------------#

#Plot the loss graph
flow.eval()
plt.subplot(211)
plt.plot(loss['train'] + np.abs(np.min(loss['train'])), label='Train')
#Loss has to have the minimum added because otherwise on a log scale it goes wild when loss
plt.plot(loss['val']+ np.abs(np.min(loss['val'])), label='Val.')
plt.yscale('log')
plt.xlabel('Epoch')
plt.ylabel('Loss (log(loss + |min|))')
plt.legend()

plt.subplot(212)
plt.plot(loss['train'], label='Train')
plt.plot(loss['val'], label='Val.')
#plt.yscale('log')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

#plot the Loss graph
plt.plot(loss['delta'])
plt.xlabel('Epoch')
plt.ylabel('Loss (log)')
plt.show()
```

```
#   TESTING THE AI ON REAL DATA #
#-------------------------------#
test_array = []
indices = []
N = 3 #number of graphs to predict
#take N random sets of input
for n in np.arange(N):

    i = random.randint(0,len(m1))
    indices.append(i)
    temp = random.choice(conditional)
    test_array.append(temp)

#prep test data
test_array = np.array(test_array)
cond = torch.from_numpy(test_array.astype(np.float32)).to(device)

#Take many Flow predictions to average over for all N samples
Big_Samples = []
N_Samples = 100
cond = torch.from_numpy(test_array.astype(np.float32)).to(device)

#create N samples
with torch.no_grad():
    for i in np.arange(N_Samples):
        #flow.sample can take N samples quickly so we do this N_Samples times
        samples  = flow.sample(N,conditional = cond)
        Big_Samples.append(samples)

#Get the samples in a workable form for cpu
for i in np.arange(len(Big_Samples)):
    Big_Samples[i] = Big_Samples[i].cpu().numpy()



Big_Samples = np.array(Big_Samples)

final_samples = np.mean(Big_Samples,axis = axis)

#create a region of 3 standard deviations.
std = np.std(Big_Samples,axis = axis)
max_lines = final_samples + 3* std
min_lines = final_samples - 3* std


cond = cond.cpu().numpy()

cmap =cm.get_cmap('viridis') #so we can set the colour of all the plots
```

```
for n in np.arange(N):
    col = cmap(n/N) #cmap takes inputs 0->1

    #generate and plot lightcurves using the model (The DU17 model imported at top)
    m1_,m2_,l1_,l2_ = cond[n]
    lc = Generate_LightCurve(m1_,m2_,l1_,l2_)[1]
    lc = np.nan_to_num(lc)
    plt.plot(lc[0],lc[1][bandindex],"--",label = f"[{m1_:.3g}, {m2_:.3g}, {l1_:.3g}, {l2_:.3

    #plot the flow predictions
    plt.plot(t_d,scaling_constant*final_samples[n],"-",ms =4,c = col)
    plt.fill_between(t_d,min_lines[n]*scaling_constant,max_lines[n]*scaling_constant,alpha =

plt.gca().invert_yaxis()
plt.legend()
plt.show()

#           SAVE THE MODEL      #
#-------------------------------#
#   This method of saving needs #
#   work, didn't really work for#
#   me.                         #

models = os.listdir("Models/")
print(models)
i = 0
for m in models:
    if m == f'model_{i}_{band}.pth' :
        print(m," Already taken")
        i += 1
    else:
        torch.save(flow,f"Models/model_{i}_{band}.pth")
        print(f"Model saved as Models/model_{i}_{band}.pth")
torch.save(flow,f"Models/model_{i}_{band}.pth")
print(f"Model saved as Models/model_{i}_{band}.pth")
```

References

[1] Abbott and et al 2017 *Physical Review Letters* **119**(16) URL https://doi.org/10.1103/PhysRevLett.119.161101

[2] Abbott and et al 2017 *The Astrophysical Journal Letters* **850**(L39) URL https://doi.org/10.3847/2041-8213/aa9478

[3] Dietrich T and Ujevic M 2017 *Classical and Quantum Gravity* **34**(10) URL https://doi.org/10.1088/1361-6382/aa6bb0

[4] Metzger B D 2017 *Living Reviews in Relativity* **20**(3)

[5] Wallace T 2022 Generating kilonova light curves using normalised flow machine learning Tech. rep. Glasgow, Scotland

[6] Menezes D P 2021 *Universe* **7**(8) URL https://doi.org/10.3390/universe7080267

[7] Oppenheimer J R and Volkoff G 1939 *Physical Review* **55**(4) 371–381 URL https://journals.aps.org/pr/pdf/10.1103/PhysRev.55.374

[8] Carroll S M *"Spacetime and Geometry"* (Cambridge University Press)