



UNIVERSITY OF  
BIRMINGHAM

---

Fusion Neutron Activation Spectra Unfolding by Neural Networks  
(FACTIUNN)

---



author: Ocean Wong  
(Hoi Yeung Wong)

supervisor: Ross Worrall

Submitted in fulfilment of the requirement for: MSc. Physics and Technology of Nuclear Reactors

date: June-September 2019

student ID: 1625143

---

I warrant that the content of this dissertation is the direct result of my own work and that any use made in it of published or unpublished materials is fully and correctly referenced.

**Abstract**

Include these things:

- Attempted this
- Got this result
- advice for the future

*Keywords:* activation, neutronics, fusion

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Theory</b>	<b>5</b>
2.1	General unfolding methods . . . . .	6
2.2	Current practice . . . . .	7
2.3	Neural Networks . . . . .	7
2.3.1	Forward Propagation . . . . .	7
2.3.2	Backpropagation . . . . .	8
2.3.3	Universal approximation theorem . . . . .	9
2.3.4	Training the neural network . . . . .	10
2.3.5	Applying neural network to the unfolding problem . . . . .	10
<b>3</b>	<b>Literature review</b>	<b>12</b>
<b>4</b>	<b>Proof of concept on simulated spectra</b>	<b>13</b>
4.1	Fully determined case . . . . .	13
4.2	Underdetermined case . . . . .	14
<b>5</b>	<b>Neural networks trained on real spectra</b>	<b>17</b>
5.1	Hyperparameter Optimization . . . . .	18
5.2	Results of predicting using the real spectra . . . . .	20
5.3	Benchmarking against existing codes . . . . .	25
5.3.1	As an unfolding tool . . . . .	25
5.3.2	As an a priori generator . . . . .	27
5.4	An attempt at using fission data to predict fusion data . . . . .	27
<b>6</b>	<b>Potential Future improvements</b>	<b>28</b>
<b>7</b>	<b>Conclusion</b>	<b>29</b>
	<b>Appendices</b>	<b>33</b>
<b>A</b>	<b>Neural network building functions tailored for the purpose of neutron spectrum unfolding</b>	<b>33</b>
<b>B</b>	<b>Neural network abstractions and controller</b>	<b>59</b>
<b>C</b>	<b>Code for benchmarking</b>	<b>65</b>
<b>D</b>	<b>Fully determined simulation data generation</b>	<b>66</b>
<b>E</b>	<b>Underdetermined simulation data generation</b>	<b>67</b>
<b>F</b>	<b>Training and evaluating neural networks on the underdetermined simulation data</b>	<b>72</b>
<b>G</b>	<b>Selecting from UKAEA and IAEA compendium</b>	<b>73</b>
<b>H</b>	<b>hyperparameter input controller</b>	<b>77</b>
<b>I</b>	<b>hyperparameter optimization searching</b>	<b>80</b>
<b>J</b>	<b>Loss value visualizer</b>	<b>84</b>
<b>K</b>	<b>Parametrisation of the FISPACT reference spectra</b>	<b>86</b>

## List of Figures

1	Illustration of the topology of a typical neural network . . . . .	7
2	A ReLU function (a rectifying function) . . . . .	8
3	A cubic function approximated by a neural network . . . . .	9
4	The spectra predicted by a 2-hidden-layers neural network (with 16 nodes per layer) on a fully determined system. . . . .	14
5	Data augmentation performed to create simulated spectra. . . . .	16
6	An example of the neutron spectrum as unfolded by the predicting a perturbed JET first wall spectrum. . . . .	17
8	All fusion spectra used, obtained from [37] . . . . .	18
7	Microscopic cross-section of each reaction . . . . .	18
9	Heatmap visualizing the loss values of the nerual networks' prediction on the test dataset. . . . .	21
10	A typical 0-hidden-layer neural network prediction of JET's first wall spectra when trained on fusion spectra (learning rate=0.01) . . . . .	22
11	A typical 1-hidden-layer neural network's prediction of JET's first wall spectra when trained on fusion spectra(learning rate=0.01) . . . . .	22
12	A typical 2-hidden-layer neural network prediction of JET's first wall spectra when trained on fusion spectra (learning rate=0.01) . . . . .	23
13	JET first wall spectrum as predicted by the optimally performing NN among all NN trained on fusion data. . . . .	23
14	Water-cooled ceramic breeder blanket TOKAMAK's first wall spectrum as predicted by the optimally performing NN among all NN trained on fusion data. . . . .	24
15	Helium-cooled LiPb TOKAMAK's spectrum at the vacuum vessel, as predicted by the optimally performing NN among all NN trained on fusion data. . . . .	24
16	Frascati neutron generator spectrum, as predicted by the optimally performing NN among all NN trained on fusion data. . . . .	25
18	JET first wall spectrum as unfolded by GRAVEL upon using a flat neutron spectrum as the <i>a priori</i> . . . . .	26
17	JET first wall spectrum as unfolded by MAXED upon using a flat neutron spectrum as the <i>a priori</i> . . . . .	26
19	JET first wall spectrum as unfolded by GRAVEL upon using the optimal NN's output as the <i>a priori</i> spectrum. . . . .	27
20	(calculated) ITER spectrum as predicted by the optimal NN among all NN trained on fission spectra . . . . .	28
21	JET first wall spectrum as predicted by the optimal NN among all NN trained on fission spectra . . . . .	28

# 1 Introduction

In a fusion reactor, the neutron fluence can go up to as high as  $1.7 \times 10^{21} \text{cm}^{-2}$  [18]. This leads to an unprecedented need of shielding against neutrons of up to 14.1 MeV or higher energies.

Neutrons are notoriously difficult to shield against due to their uncharged nature, and therefore low propensity to interact with matter [citation needed]. To develop effective shielding for various components of the reactor from these high energy neutrons, the energy spectrum of the neutrons created inside the nuclear reactor has to be well understood [8].

It is also important to understand the neutron spectrum inside the reactor in order to develop Tritium breeding modules; a Tritium Breeding Ratio  $\text{TBR} > 1$  is essential for making fusion a sustainable source of clean energy [20]. Last but not least, the power output of future fusion power plants can only be quantified when the neutron spectrum is characterised [46]. An accurate measurement of the neutron spectrum is required to properly model the energy distribution of neutrons to be used in neutron transport simulations for the above purposes.

All of the above activities relies on having accurate knowledge of the neutron spectrum. Therefore neutron spectroscopy is a key focus in the diagnostic systems in all fusion reactors.

Incidentally, for the same reason that they are difficult to shield against, neutron energy is also difficult to measure. Neutrons, especially high energy neutrons such as the 14.1 MeV neutrons created in fusion reactors, do not easily deposit their full energy into a sufficiently small detection volume to allow direct measurement [21]. Various neutron detectors has been developed to deal with this problem; however, most of them cannot stand this high neutron fluence at the first wall of fusion reactors without additional shielding that changes the flux profile [41]. The extreme temperature and magnetic fields inside the nuclear fusion reactor compounds the difficulty of employing other means of neutron measurement as most electronics will not be able to function in such environments effectively. [30]

This is where the technique of activation foil unfolding stands out: By analyzing the level of activations in various elements induced by neutrons, relying on the variation of reaction cross-sections across different elements and energies, one can infer the neutron spectra that was previously present at the first wall.

This is a very robust method as it does not require any active components, thus can be employed for very high neutron fluxes [10], as the total number of neutron activation reactions can be controlled by changing the volume of the activation foils used [12] according to the anticipated neutron fluence in the next irradiation period, so not to paralyze the  $\gamma$  radiation detector. It is also insensitive to  $\gamma$  rays, thus removing most of the challenges facing mixed-field spectrometry. [5]

The disadvantage of this method is that it has to be time-integrated over the whole irradiation period, thus no information about the temporal variation in the neutron spectrum can be gleaned from its measurements.

Another disadvantage of using neutron activation as the means of measuring the neutron spectrum in a fusion reactor is that it is an indirect method of measurement, requiring the measured reaction rates to be ‘unfolded’ back into reaction rates. This is a ‘mathematically incorrectly posed’ problem [22], as will be further explained in the next section (2.3), requiring an initial guess spectrum to be provided before the unfolding procedure can take place. This is because the number of reaction channels recorded (usually denoted as  $M$ ) is fewer than the number of neutron groups (usually denoted as  $N$ ) whose neutron flux we would like to know, i.e.  $M < N$ , thus this problem is underdetermined (the number of constraints is fewer than the number of variables). The *a priori* has to be used in order to introduce extra information into the problem. However, if this *a priori* spectrum deviates too much from the actual spectrum, then the result of the unfolding will be inaccurate.

To address this problem, an investigation into using neural networks for the purpose of unfolding fusion neutron spectra is presented in this thesis. Neural networks excels in incorporating previous spectra as *a priori* information, without requiring users to explicitly input an *a priori*. Two approaches are proposed. The first one is to use neural networks directly as an unfolding tool; and the second one is to use them as an *a priori* generator, which is then fed into an existing unfolding code, where the actual neutrons spectra is then calculated out of.

## 2 Theory

When a nuclide is placed in the activation module at the irradiation position inside a nuclear fusion reactor (or any other neutron sources), it is activated via one or more nuclear reactions with the incoming neutrons. The probability of interacting with the incoming neutron via reaction  $j$  is proportional to the microscopic cross-section  $\sigma_j(E)$ , where  $E$  is the neutron's energy, and reaction  $j$  is a neutron-induced reaction, i.e. (n,*any*) reaction.

By measuring the activity of reaction  $j$ 's daughter nuclide in the activation foil (which has a known amount of the initial nuclide) after irradiation, and multiplying it by a correction factor of

$$\frac{1}{1 - \exp(-\lambda_j T)} \quad (1)$$

the reaction rate  $Z_{0j}$  can be obtained. This correction factor accounts for the decay of the daughter nuclide of reaction  $j$  which has a half-life of  $\lambda_j$ , over the period  $T$  which is the duration between irradiation and measurement. A more complicated correction factor is required if the irradiation period is comparable to the half-life  $\lambda_j$ , or if the population of the parent nuclides for reaction  $j$  changes over the course of the irradiation. This can be done using FISPACT-II, detailed in [38].

The total reaction rate of the  $j^{th}$  reaction can then be expressed as a Fredholm integral as follows:

$$Z_{0j} = \int_0^\infty R_j(E) \phi_0(E) dE \quad (2)$$

where the reaction rate  $Z_{0j}$  has the unit of  $s^{-1}$ ,  $\phi_0$  is the neutron flux (unit:  $cm^{-2}s^{-1}$ ), which is a function of energy  $E$ . The unfolding process aims to find a solution spectrum  $\phi$  which approximates the actual spectrum  $\phi_0$  as closely as possible.

As for  $R$  in the equation above, (which has dimension of area)

$$R_j(E) = \sigma_j(E) \frac{N_A}{A} F_j \rho V \quad (3)$$

assuming that there is no self-shielding/down-scattering inside the foil.  $N_A$  is the Advogadro's constant (unit:  $mol^{-1}$ ),  $A$  is the molar mass of the parent nuclide for reaction  $j$  (unit:  $g\ mol^{-1}$ ),  $F_j$  is reaction  $j$ 's parent isotope's mass fraction in the foil's constituent material (unit: dimensionless),  $\rho$  is the density of the alloy (unit:  $g\ barn^{-1}\ cm^{-1}$ ),  $V$  is the volume of the foil (unit:  $cm^3$ ) Note that  $\sigma(E)$  (unit:  $barn$ ) is the only energy dependent component in  $R$ .

The neutron spectrum can be discretized into  $N$  energy bins:

$$Z_{0j} = \sum_{i=1}^N R_{ji} \phi_{0i} \quad (4)$$

where  $\phi_{0i}$  is the scalar flux integrated over the energy bin's range

$$\phi_{0i} = \int_{E_{i-1}}^{E_i} \phi_0(E) dE \quad (5)$$

, thus having a unit of  $cm^{-2}s^{-1}$ .

By assuming that the scalar flux distribution inside each energy bin is relatively flat, equation 4 calculates  $Z_{0j}$  by replacing  $(R_j(E), E_{i-1} \leq E \leq E_i)$  with

$$R_{ji} = R_j(E_{i-1}) \quad (6)$$

Let there be  $M$  neutron-induced reactions whose reaction rate was measured,

$$\begin{aligned} \forall j \in \{1, \dots, M\}, \\ \exists Z_{0j} \in \mathbb{R}_{\geq 0} \end{aligned} \quad (7)$$

Collecting all reaction rates into a vector  $\mathbf{Z}_0$  of  $M$ -dimensions, one can express eq. 4 as a matrix multiplication equation:

$$\mathbf{Z}_0 = \underline{\underline{\mathbf{R}}}\phi_0 \quad (8)$$

where  $\underline{\underline{\mathbf{R}}}$  is a  $M \times N$  matrix, termed the *response matrix*.  $\phi_0$  is an  $N$ -dimensional vector containing the neutron flux in the each of the  $N$  bins. The subscripts 0's denotes that they are the measured/known quantity, as opposed to the conjectured solutions which will appear later in this text.

For nuclear fusion applications, the number of possible reaction investigated  $M$  is very limited [24], as the parent nuclide of each of these reactions must exist in solids which:

- can be manufactured into specified shape and thickness, with well measured number density and impurity contents,
- are safe to be handled,
- has a threshold energy in the region of interest (in the MeV range),
- has well-characterised cross-section values in nuclear data libraries (see [11])
- has stable parent isotope and daughter isotopes of medium length half-lives such that it can be activated and measured.

in practice, very few types of metals/alloys can be used in these systems. For the ACT in JET in particular, in recent experiments, only 7 types of foil materials and 11 reactions were examined. [38]

Meanwhile, the number of bins,  $N$ , can be arbitrarily high. For some investigations, such at the one in [36] it goes up to 709 bins. This makes the unfolding problem a strongly underdetermined one.

In the mathematical sense of the problem, an inverse does not exist. This is because, theoretically, multiple neutron spectra, say  $\phi_0$ ,  $\phi_1$  and  $\phi_2$ , can give the same set of reaction rates  $\mathbf{Z}_0$ , so there is no correct, unique choice of mapping of  $\mathbf{Z}_0$  back to  $\phi_0$ ,  $\phi_1$  and  $\phi_2$ .

Such an inverse problem is termed ‘mathematically incorrectly posed’. [22]

## 2.1 General unfolding methods

The most straight-forward way of getting back a solution  $\phi$  is by using the Moore-Penrose inverse matrix. This matrix inversion operation generalizes the usual matrix inversion operation for square matrices, where the  $M \times N$  response matrix  $\underline{\underline{\mathbf{R}}}$  in equation 8 is inverted into an  $N \times M$  matrix  $\underline{\underline{\mathbf{R}}}^{-1}$ , so that  $\phi$  can be obtained by  $\phi = \underline{\underline{\mathbf{R}}}^{-1}\mathbf{Z}_0$ . However, this method is the equivalent of rotating a 2-D photo of a 3-D object from a horizontal position to an upright/tilted position: the solution is still “trapped” in a flat,  $M$ -dimensional manifold within the  $N$ -dimensional solution space.

Therefore to start the unfolding process, extra information has to be given to the program. This is termed the *a priori* spectrum.

The most general unfolding program can, ideally, find a solution  $\underline{\mathbf{Z}}$ ,  $\underline{\mathbf{R}}$  and  $\phi$  [27], such that their overall deviation from the measured reaction rates ( $\mathbf{Z}_0$ ), expected response matrix ( $\mathbf{R}_0$ ), and the initial guessed neutron spectrum ( $\phi_0$ ), is minimized. The deviation of the solution reaction rates from the measured reaction rates is calculated from its covariance matrix  $\underline{\mathbf{S}}_{\underline{\mathbf{Z}}}$ , as the  $(\chi^2)_Z = \underline{\mathbf{Z}}^T \underline{\mathbf{S}}_{\underline{\mathbf{Z}}}^{-1} \underline{\mathbf{Z}}$ . Equivalently the deviation of  $\phi$  from  $\phi_0$  and  $\underline{\mathbf{R}}$  from  $\mathbf{R}_0$  can be calculated from their respective covariance matrix..

## 2.2 Current practice

In practice, there is always uncertainty associated with the cross-section values provided by the nuclear data libraries [28]. This is known as the “ambiguity” of the response matrix. To reduce the complexity of the problem, however, the ambiguity in the response matrix is nearly always ignored, by assuming that the response matrix  $\mathbf{R}_0$  is accurately and precisely defined, fixing the response matrix during the solution search. This reduce the number of dimensions in the solution search by  $M \times N$ , massively reducing the computational complexity. It also assumes that the covariance matrix of the reaction rates is diagonal, i.e. there are no covariance across different reaction rates.

Some programs, such as GRAVEL[26] and SAND-II[29], simply start their iterative solution search from this *a priori* spectrum, with the aim of minimizing the  $\chi^2$  (which measures the deviation of  $\underline{\mathbf{Z}}$  from  $\mathbf{Z}_0$ ); while others, such as MAXED [40] add the deviation of the solution spectrum from the *a priori* spectrum ( $\phi$  from  $\phi_0$ ) on top of the deviation of the solution reaction rates from the measured reaction rates ( $\underline{\mathbf{Z}}$  from  $\mathbf{Z}_0$ ) when evaluating the  $\chi^2$ .

Current fusion neutron measurements relies on MCNP simulations heavily to supplement their unfolding procedure. They use MCNP model of thre reactor to calculate a neutron spectrum, which is used as the *a priori* [25] [23]; and the response matrix is usually obtained in the same way as well [12].

## 2.3 Neural Networks

Neural networks, on the other hand, learns the relationship between reaction rates and the original neutron spectrum. Ideally it will make use of information in previous neutron spectra, effectively bypassing the problem of underdetermination.

A typical neural network learns the relationship between the inputs (the two nodes in the leftmost layer in Figure 1) and outputs (the node in the rightmost layer in Figure 1) of a function via training, thus becoming an approximator for that function.

### 2.3.1 Forward Propagation

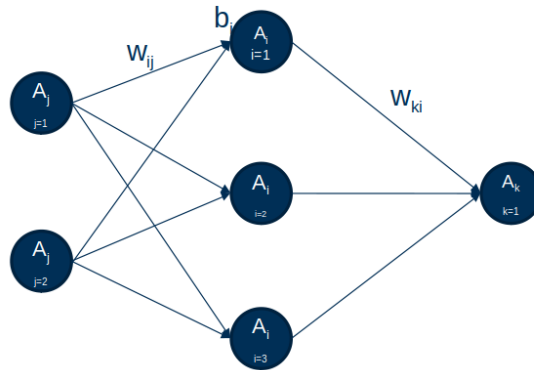


Figure 1: Illustration of the topology of a typical neural network



Figure 2: A ReLU function (a rectifying function)  
Abscissa=function’s input; ordinate=function’s output.

The inputs to the neural network are known as “features” and the outputs are known as the “labels”.

In the context of neutron spectrum unfolding using neural networks, there are  $M$  features (reaction rates  $Z_j$  for  $1 \leq j \leq M$ ) and  $N$  labels (neutron flux in each bin  $\phi_j$  for  $1 \leq j \leq N$ ).

The “activation”  $A_i$  of the  $i^{th}$  node refers to the value that it takes.  $w_{ij}$  denotes the “weight” of each connection from the  $j^{th}$  node to the  $i^{th}$ .

When the activations in the input layer ( $A_j$ ) are known, the activation in the next layer (in this case, the first hidden layer) is calculated as follows:

$$A_i = \sigma_i \left( \sum_j (w_{ij} A_j) + b_i \right) \quad (9)$$

$b_i$  denotes a “bias” value which will be added onto the sums in front of each node before it is parsed through the activation function  $\sigma_i$ . The activation function is usually denoted as  $\sigma_i$ , i.e. it is possible to use different activation functions for different nodes  $i$ ; however the common practice is to use the same type of activation function across the whole layer, or even across all nodes and all layers of the neural network. The typical function chosen is the ReLU function (Figure 2), i.e. for all layers, and for all values of  $i$ , as it is one of the simplest non-linear function whose gradient can be computed quickly.

$$\sigma_i(x) = ReLU(x) = \frac{|x| + x}{2} \quad (10)$$

Equation 9 is applied recursively to calculate the activations in the immediate next layer. For example, to calculate the activations in second layer (i.e. the output layer) in Figure 1 simply by swapping the indices in for the indices of the next layer:  $i \mapsto h, j \mapsto i$ . This process is known as forward propagation.

### 2.3.2 Backpropagation

The weights  $w$  and biases  $b$  are known as the parameters of the neural network. This is in contrast with the term “hyperparameters”, which are the numbers that describes the topology of the neural network, i.e. number of layers, number of nodes in each layer, learning rate (see section 2.3.4 below), etc. During the training phase of the neural network, these parameters are adjusted so that the neural network’s predicted output values align with the true output values more closely. This deviation of the predicted label from the true label is termed the “loss value”, and can be calculated in a variety of manner (see Section 11 for the loss value metrics considered in this investigation). For the moment let’s assume it is calculated as the mean-squared value, i.e. same as the  $\chi^2$  value familiar to physicists.

The process of adjusting parameters to reduce the loss value is known as backpropagation, as the ‘desired’ change to each weight and bias (calculated from the gradient of the



loss value with respect to  $w$  or  $b$ , i.e.  $\frac{\partial(loss)}{\partial w}$  or  $\frac{\partial(loss)}{\partial b}$ ) is obtained by tracing the change in the output layer back to the weight and biases of each layer.

For the neural network to converge on a stable set of parameters (i.e. a minimum value of the loss value in the parameter space), features are usually normalized before they are given to the neural network. This reduces the difference in variance across each feature, allowing the neural network to take a more direct path when gradient-descending to the set of parameters that achieves minimum loss value, instead of an oscillatory approach to the minimum loss value[33], thus reducing the number of steps required to train the neural network.

### 2.3.3 Universal approximation theorem

Before diving into the details of neural network training, it is beneficial to see how a neural network can approximate any function.

The key to its ability of approximating functions lies in the non-linear activation function.

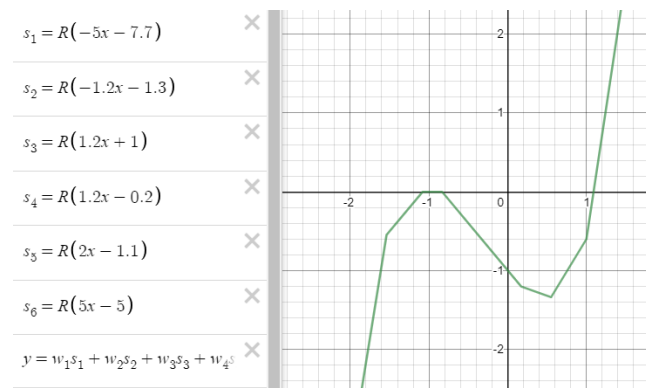


Figure 3: A cubic function approximated by a neural network

This neural network has 1 hidden layer containing 6 neurons. Here,  $R$  is the alias for the ReLU function (see Figure 2), abscissa is the input layer neuron's activation value (feature); and the ordinate is the output layer neuron's activation value (label), obtained by summing over the product of the activation of the  $i^{th}$  neuron (aliased as  $s_i$ ) with the weight of its connection to the final layer (aliased as  $w_i$ ). The weights and bias to the first layer is already defined on-screen inside the brackets of the first six lines; while the weights and bias to the second layer (output layer) is defined off screen.

Figure 3 is a crude representation of how a 1-hidden-layer neural network can approximate a cubic function. A single hidden layer neural network with one scalar input and one scalar output is able to approximate any non-linear functions, provided that there are enough neurons in the hidden layer.

The weights  $w_i$  scales each ReLU function; whereas the bias to the first layer (defined as the second term inside each of the bracket in the first six lines) changes the horizontal offset of each ReLU function. The bias to the second layer controls the vertical offset of the whole function. Summing them up leads to the output in Figure 3.

Even with only six hidden neurons (only 19 parameters), it is able to reasonably approximate a cubic function within the visualized domain. Obviously this approximation to the cubic function can be improved by increasing the number of neurons available in the neural network, provided that there are enough training data to cover the domain densely.

Armed with this intuition, the notion of a neural network being able to approximate any function becomes conceivable, even if the function has vectorial inputs and outputs.

### 2.3.4 Training the neural network

Before adjusting the parameters, a fraction of the data is drawn out and reserved for testing. The remaining is used as the training dataset. These “training” and “testing” data are chosen in such a way that they cover the same range of domain and co-domain in the feature- and label-space respectively.

The parameters are adjusted iteratively to minimize the loss value. Each step requires calculation of the gradient value over the entire dataset, known as an “epoch”, obtained by calculating the average  $\frac{\partial(loss)}{\partial w}$  or  $\frac{\partial(loss)}{\partial b}$  over the entire training dataset. The parameters are adjusted according to an algorithm known as the “optimizer”; this algorithm may require some more hyperparameters, such as the “momentum”, “acceleration” term to be defined. When further training no longer improves the loss value over the training set (i.e. the “training loss”), training can be stopped, and the parameters are then fixed at their final values. The size of each step is calculated as (learning rate)  $\times$  (gradient of the loss value in parameter space)

The performance of the neural network is then evaluated over the testing set to obtain an average loss value, known as the “testing loss”. If the testing loss is much higher than the training loss, it signifies that the neural network was “overfitting”, i.e. it reached a minimum training loss by “memorizing” the relationship between training features and training labels, and is unable to generalize these relationships to the testing set. This may suggest that the neural network is too complex, i.e. has too many nodes or neurons.

Apart from reducing the complexity of the model, various techniques exist to reduce overfitting, including weight-regularization and dropout [2]. Weight regularization ensures that the numerical values of weights  $w$  remains small; while dropout effectively removes a specified fraction of the connections at each layer. However, these techniques are not applied.

However, one of the most widely used method of reducing overfitting is by measuring the validation loss. A small subset of the training data is reserved and not used for backpropagation during the training, but its loss value (known as the “validation loss”) is calculated at each epoch also. This amounts to calculating the loss value of the neural network’s prediction on a set of data that it has never seen before as well. When the validation error stops decreasing, then one can be sure that the neural network has stopped identifying general patterns which applies across both the validation set and the training set, and begin memorization. The training can be stopped at this point.

This method is called “Early Stopping”; it catches the neural network before it begins overfitting aggressively.

### 2.3.5 Applying neural network to the unfolding problem

To apply neural networks as unfolding tools, we will want neutron spectra as the output and reaction rates as the input, i.e.  $M$  features (reaction rates  $Z_j$  for  $1 \leq j \leq M$ ) and  $N$  labels (neutron flux in each bin  $\phi_j$  for  $1 \leq j \leq N$ ).

By using a neural network to do the unfolding, we are assuming that the inverse equation (below) exist,

$$\underline{Z} = \underline{\underline{R}}^{-1} \phi \quad (11)$$

i.e. all reaction rates can be unfolding back to one and only one unique solution spectrum. Ideally, the set of all possible solution for the neutron spectrum  $\phi$  can be expressed as a linear sum of  $M$  or fewer basis vectors due to the constraints of various physical processes. The role of the neural network is to identify these  $M$ - bases vectors, and relate them to the  $M$  reaction rates.

Several metrics were considered for the neural networks. Since the neural networks’ goal is to predict a set of labels (solution spectrum)  $\phi_{pred}$  that is identical to the true

spectrum  $\phi_0$  when given the set of features  $Z_0$  corresponding to the set of labels  $\phi_0$ , the loss function must have a minimum at  $\phi_{pred} = \phi_0$ .

This loss value is also expected to scale its penalization according to the true flux  $\phi_0$ . Large deviation when  $\phi_0$  is large should be penalized by the same amount as with small deviations when  $\phi_0$  is small. For example, over-predicting the to flux at the 14.1 MeV peak by, say, 10%, in a DD-operation, should be given the same penalty as over-predicting the 14.1 MeV peak flux in a DT-operation by 10%, despite the fact that  $\phi_0(E = 14.1 \text{ MeV})$  is much smaller for the same TOKAMAK in a DD campaign than in a DT campaign.

Several of such functions comes to mind; they include:

- cross entropy,  $H(\phi_{pred}, \phi_0) = \sum_i^N \left( \phi_{pred}(E_i) (\ln(\phi_{pred}(E_i)) - \ln(\phi_0(E_i))) \right)$  (See [43])
- Average distance in  $L^P$  log-space  $= \left( \sum_i^N (\log(\phi_{pred}) - \log(\phi_0))^p \right)^{\frac{1}{p}}$  which is a generalization of mean squared error and mean absolute error.
- mean fractional deviation,  $MFD(\phi_{pred}, \phi_0) = \sum_i^N \left| \frac{\phi_{pred}(E_i) - \phi_0(E_i)}{\phi_0(E_i)} \right|$

In the end, the following functions were chosen as they were the default functions available from tensorflow; using these functions minimizes the room for human error and development time.

Let there be  $L$  features-labels pairs in the dataset. The loss values are defined as:

- mean squared error:

$$MSE(\phi_{pred}, \phi_0) = \frac{1}{L} \sum_k^L \sum_i^N \left( \log_{10}(\phi_{pred,k}(E_i)) - \log_{10}(\phi_{0,k}(E_i)) \right) \quad (12)$$

- mean pairwise squared error:

$$MPSE(\phi_{pred}, \phi_0) = \frac{1}{L} \sum_k^L \sum_i^N \sum_q^N \left( \log_{10} \left( \frac{\phi_{pred,k}(E_i)}{\phi_{pred,k}(E_q)} \right) - \log_{10} \left( \frac{\phi_{0,k}(E_i)}{\phi_{0,k}(E_q)} \right) \right) \quad (13)$$

The neural networks in this investigation differ from the typical neural network, in that the latter has fewer labels than features output ( $N \leq M$ ); and that, since the features is related to the labels via a physical process, the inverse function for turning labels back into features exist (equation 8), and is assumed to be deterministic.

This allows for an additional information to be supplied to the neural network during the training stage:

- mean squared error including folded reaction rates:

$$MSE_{\text{including\_folded\_reaction\_rates}} = MSE(\phi'_{pred}, \phi'_0) \quad (14)$$

- mean pairwise squared error including folded reaction rates:

$$MPSE_{\text{including\_folded\_reaction\_rates}} = MPSE(\phi'_{pred}, \phi'_0) \quad (15)$$

Where  $\phi'$  is the  $\phi$  and  $Z$  vector concatenated together,

$$\phi' = [\phi_1, \dots, \phi_N, Z_1, \dots, Z_M] \quad (16)$$

and  $Z$  is, in turn, obtained by equation 8:

$$Z_{pred} = \underline{\underline{R}} \phi_{pred} \quad (17)$$

$$Z_0 = \underline{\underline{R}} \phi_0 \quad (18)$$

This is analogous to the technique of regularization[13] in normal unfolding procedures, where both deviation from the *a priori* spectrum and the reaction rates are calculated and used as the  $\chi^2$  value. In this case, the regularization constant (weight of the neutron flux's deviation relative to the reaction rates' deviation) is simply chosen as 1.

These two metrics will give loss value = 0 when  $\phi_{pred}$  and  $\phi_0$  matches perfectly; but the neural network will be penalized by an additional amount if it makes a mistake in the spectrum that leads to a greater deviation of the  $Z_{pred}$  from the  $Z_0$  (which is a mistake that other linear/non-linear least-square unfolding codes such as MAXED and GRAVEL will not make. This effectively incorporates some physics into the neural network with the hopes of improving its accuracy.

### 3 Literature review

There are no previous attempts of unfolding fusion neutron spectra using neural networks. Therefore this technique is entirely new and not applied.

Some work has been carried out in the field of neutron spectrum unfolding using neural networks. Only one of them are directly related to the method of activation foil neutron spectrum unfolding [19], which has a more pathological response matrix than the other two methods typically discussed in unfolding (Bonner Spheres and liquid scintillators). The condition number of the response matrix (i.e. the ratio of the maximum to minimum singular value[31]) is likely worse due to the similarities between reaction cross-sections as dictated by nuclear physics; unlike in the other two detectors, where the response matrix is almost guaranteed to be lower-triangular matrix, where the response of each Bonner sphere/ scintillation pulse height is guaranteed to be linearly independent from each other, so that the condition number is to be small, i.e. they are less ill-conditioned than the problem of activation foil neutron spectrum unfolding. Therefore unfolding these two types of spectra are not as difficult as unfolding neutron spectra from activation foil.

For the purpose of neutron spectrum measurement in a nuclear fusion reactor, which has very high neutron and  $\gamma$  fluence compared to fluences typically found in medical physics and in fission reactors where Bonner spheres and liquid scintillator neutron measurement and unfolding are used. These two methods are unsuitable for measuring fusion neutron energies directly due to their low radiation tolerance relative to the method of activation foil. However, this does not mean the work of neural network unfolding of neutron spectrum from Bonner Spheres measurements [35] [34] [15] [9] and liquid scintillator measurements [17] are not useful for fusion neutron spectra unfolding. They provide useful references of neural network topologies useful for neutron spectrum unfolding (Table 1).

They also offer some insight as to how to overcome data the challenge of data scarcity. Namely, with Radial Basis Function Neural Networks (RBFNN) and General Regression Networks (GRNN), which are subsets of Probabilistic Neural Networks (PNN). They are more intricately designed than the typical Feedforward Neural Network (FNN), which is the type of neural network that has been discussed in this thesis so far. But neutron spectra unfolding with RBFNN[6][51] and GRNN[50][14][51] are more complicated than unfolding, so in thesis only FNN will be investigated in details; further investigation into using RBFNN and GRNN may follow from this thesis, in an attempt to improve fusion neutron spectrum unfolding performance.

In addition to having a more under-determined matrix than all of the cases displayed in Table 1 (number of input nodes = 11; number of output nodes = 175), the problem of unfolding fusion neutron spectra comes with the lack of data: There are very few existing nuclear fusion facilities in the world, many of which do not have a first-wall similar to JET, ITER or DEMO, where tritium is expected to be bred, and neutron spectra measurement is paramount. Since the first wall condition will affect the plasma physics and the neutron scattering greatly, very few existing fusion neutron spectra are useful for training the neural network. They mainly consist of measurements from JET

Source	topology of NN	comment
[35]	7:10:75	optimum: momentum =0.1, learning rate= 0.1, activation function = trainscg
[34]	7:14:31	optimum: learning rate= 0.1, optimal momentum = 0.1, activation function = trainscg (same author as [35])
[15]	6:10:16:6	Fully determined system
[17]	1 input layer : 2 hidden layer : 1 output layer	Fully determined system
[7]	50:50:1	over-determined (for fluence estimation)
[9]	10: 50: 52	used for unfolding monoenergetic and continuous spectra

Table 1: Topology of all of the feed-forward neural networks used for neutron spectra unfolding found in literature.

and modelling results from ITER. (This will be discussed further in Section 5.)

Some of these authors[51] have also attempted to unfold neutron spectra via other artificial intelligence methods, such as Genetic Algorithm (GA). There are some interest on this topic[45][32], but recent work here in CCFE has shown that GA is unpromising for the purpose of unfolding fusion neutron spectra [52]. For completeness, other AI methods that were considered for the purpose of neutron spectra unfolding includes Particle Swamp [42] and Artificial Bee Colony [44]. None of these methods will be considered in detail as it is beyond the scope of this thesis.

## 4 Proof of concept on simulated spectra

To demonstrate that the neural network is able to unfold neutron spectra at all, simple synthetic neutron spectra were created and folded through a response function, and neural networks were used to unfold them. These synthetic neutron spectra are simple in the sense that there are no physical processes modelled/considered while creating them, thus cannot be used to represent real world data.

### 4.1 Fully determined case

A square response matrix consisting of  $5 \times 5$  randomly picked numbers (uniformly distributed across  $1 \leq R_{ji} \leq 50$ ) was generated.

A set of 100 spectra, each containing 5 randomly picked numbers (uniformly distributed across the range  $1 \leq \phi_i \leq 15$ ) were also generated. They are regarded as the “true” neutron flux distributions. The “true” reaction rates corresponding to each spectrum was obtained by folding it through the response matrix according to equation 8. These form the features and labels respectively.

A single neural network with 0-hidden layer was able to predict the remaining labels (i.e. labels in the testing set) from the features perfectly after 10000 epochs of training over half the dataset (i.e. the training set consist of the first 50 features-labels pair). Note that at this stage, the neural network has not logarithmized the features’ or the labels’ numerical values in its pre-processing step, i.e. its loss function calculates the deviation in linear-space instead of log-space in equation 12, and regressing on the original value of the features, instead of  $\log(\text{features})$ . This was done since the features are uniformly distributed within the same magnitude, so feature normalization is not needed.

The perfect replication of the results is an expected and trivial result, as a 0-hidden-layer neural network is merely a matrix multiplication equation. Further examination of

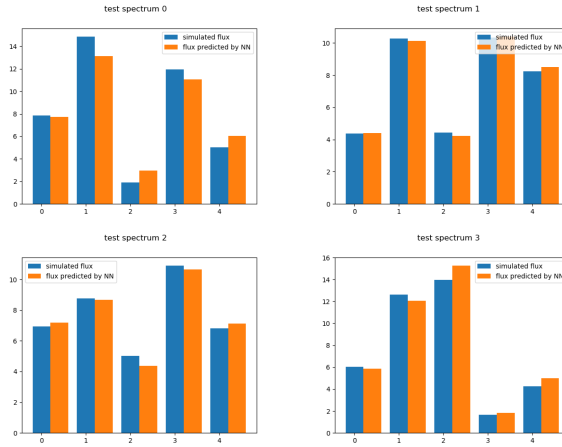


Figure 4: The spectra predicted by a 2-hidden-layers neural network (with 16 nodes per layer) on a fully determined system.

Abscissa: neutron bin number; ordinate: neutron flux (arb. units)

The loss function used is as defined in equation 12.

the weights (by enabling `eager_execution_mode` in tensorflow before re-training the neural network) shows that the weights connecting the input to the output layer forms a matrix that is identical to the transpose of the inverse matrix  $\underline{\mathbf{R}}^{-1}$  up to 3 significant figure. The difference after the 3<sup>rd</sup> (or more) significant figure were attributed to rounding errors, and the fact that the learning rate (i.e. Adam Optimizer's default learning rate of 0.001) was too big for the neural network to settle into the minimum in the parameter space properly.

The idea of logarithmizing the numerical values of features and labels in the pre-processing step were then introduced and tested on this dataset.

Since logarithms destroys the linearity of this problem, two hidden layers were added to account for the increased complexity of the problem. (Preliminary experimentation showed that adding only one hidden layer is insufficient, as the loss value does not go down as far as with the two hidden layer neural network.)

The neural network was still able to reproduce the original spectra with a somewhat satisfactory level of accuracy after training for 10000 epoch at a learning rate=0.001 over the 50 training data. Training was done using the Adam algorithm, which is the default tensorflow algorithm. Figure 4 contains 4 example plots from the testing set, as predicted by the neural network.

As expected, larger deviations were observed when the absolute value of  $\phi_{0i}$  is high, as the loss value contribution from each  $\phi_{pred i}$  is proportional to  $\frac{\log_{10}(\phi_{0i})}{\log_{10}(\phi_{pred i})}$

## 4.2 Underdetermined case

To demonstrate that the neural network is capable of performing the unfolding procedure in an underdetermined condition, 2100 spectra were made from 7 fusion neutron spectra, and then used to train and test the neural network. The 7 spectra used are listed in Table 2.

The data were rebinned into a modified Vitamin-J group structure, where the last 4 (highest energy) bins are discarded to avoid the need of extrapolating some of the spectra beyond their recorded energy ranges.

Each neutron spectrum was parametrised as a sum of 3-7 normal and log-normal distributions(see Figure 5a). The full table of the parameters used to parametrise the spectra is listed in Table 5. The parametrisation was carried out using a parameterised code currently under development at CCFE.

Code	reactor
JAEA-FNS	JAEA Fusion Neutron Source D-T
Frascati-NG	ENEA Frascati Neutron Generator D-T
ITER-DD	Magnetic confinement fusion, ITER D-D
ITER-DT	Magnetic confinement fusion, ITER D-T
DEMO-HCPB-FW	DEMO fusion concept He-cooled pebble bed, first wall
JET-FW	Joint European Torus, first wall vacuum vessel
NIF-ignition	Inertial confinement fusion, NIF ignited

Table 2: The neutron spectra used as the starting point for creating more simulated fusion neutron spectra, obtained from [48]

Each simulated new spectrum is created using the following procedure: using the parametrised representation of one of the above spectra, 40% of these distributions in the parametrised representation were randomly selected to have their amplitudes of scaled up/down by a random factor (picked from a lognormal distribution with  $\mu = 0, \sigma = 1$ ), leaving the remaining 60% of them un-perturbed.

This process was repeated 300 times for each spectrum in Table 2, giving the 2100 spectra in Figure 5b.

The purpose of this parametrisation is such that an underlying pattern can be introduced into the spectrum, which the neural networks are expected to identify on its own during the training stage.

Two neural networks were created. An arbitrarily chosen network topology of 11:128:256:175 was used. 80% of the data were randomly selected to become the training set; while the remaining becomes the testing set. The training set is further subdivided such that 20% of it is used as the validation set (i.e. 16% of the original features-labels pair becomes the validation set). The technique of Early Stopping was applied so that if the neural network shows no improvement in validation loss in 1000 epochs, the training will be stopped and the parameters (weight and biases of every layer) will be restored to the values achieved in that epoch which has the minimum validation loss. The maximum number of epoch allowed for the training was set to 10000; however both neural networks finished training (i.e. reached a minimum validation value with no further improvement for 1000 epoch) before reaching the 10000 epochs mark.

The first neural network uses mean-squared-error as the metric for calculating loss value (equation 12); while the second uses mean-squared-error-including-folded-reaction-rates (equation 14).

Interestingly, the latter achieved a slightly lower loss value instead of the former, and finishes training earlier than the former, despite having its loss function sum over a longer vector and therefor more terms to sum over more terms to obtain the loss value. The details are shown in Table 3.

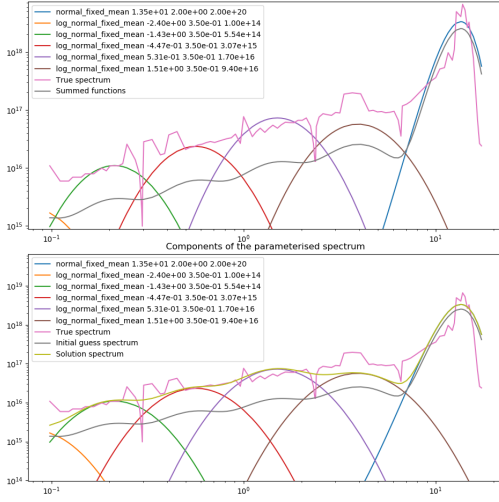
The last row in Table 3 is defined as

$$\text{std-dev-log}(C/E) = \sum_j^M \ln \left( \frac{Z_{pred,j}}{Z_{0j}} \right) \quad (19)$$

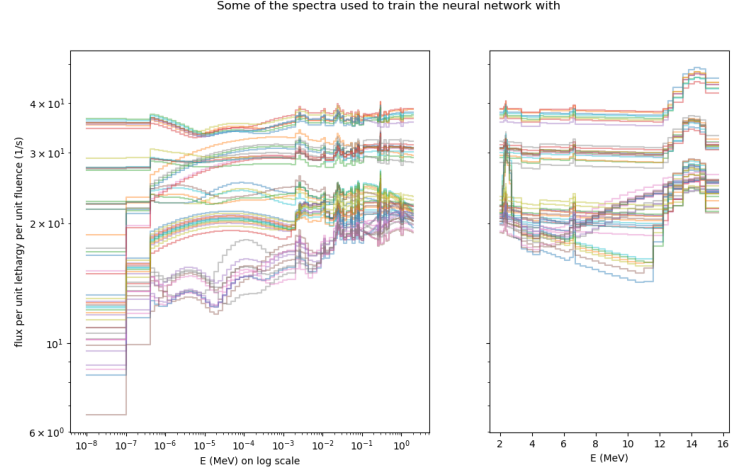
This quantity measures the sum of squares of deviation of reaction rates in log space, where  $Z_{pred}$  is obtained via equation 17.

An example of the second neural network's prediction is shown in Figure 6.

The low training loss/MAE/MSE shows that the neural networks were able to learn the relationship between the features (reaction rates) and labels (neutron spectra), and then replicate the underlying pattern in the label; while the test loss/MAE/MSE were only slightly higher than the training loss/MAE/MSE (i.e. it is within a factor of 2 of the latter), suggesting that the neural network has learnt to do so without loss of generality.



(a) An example of parametrisation performed on the the NIF spectra. These flux values are the total flux inside each energy bin, *not* divided by the lethargy span of each bin, so they are higher/lower in wider/narrower energy bins. The top plot is the initial guess parametrisation, the bottom plot is the final parametrised spectrum.



(b) 300 perturbed spectra were generated for each of the 7 original fusion spectra are plotted here, in flux per unit lethargy.

Figure 5: Data augmentation performed to create simulated spectra.

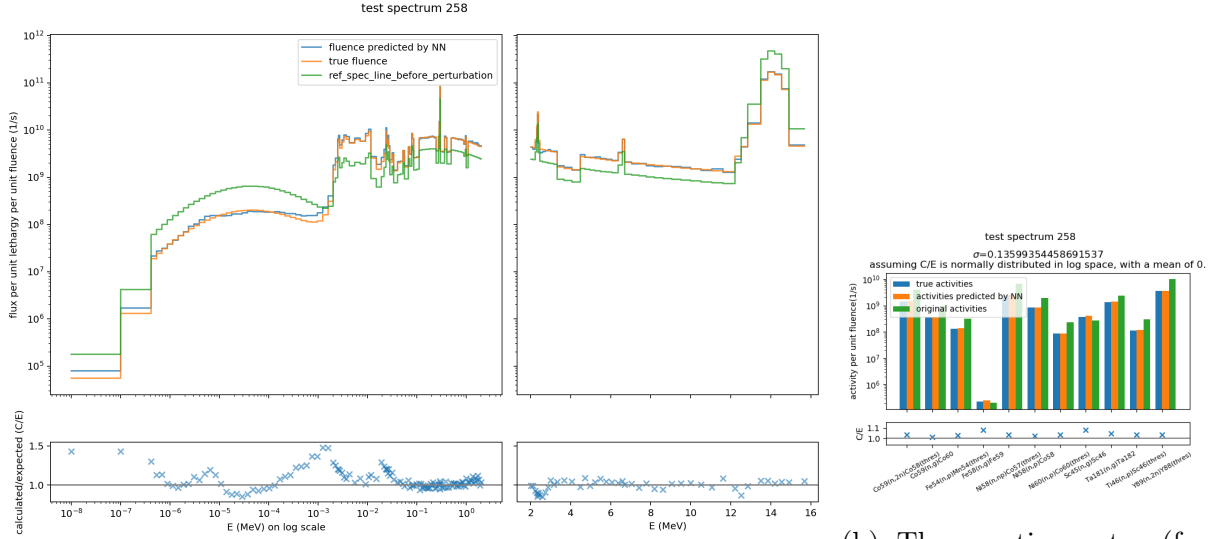
loss value	defined in eq. 12	defined in eq. 14
number of epochs of training required before a minimum val. loss is reached	6607	5078
training loss	0.29554	0.27586
training MAE	0.38083	0.37598
training MSE <sup>†</sup>	0.29554	0.29201
validation loss	0.34390	0.30682
validation MAE	0.37107	0.36681
validation MSE <sup>†</sup>	0.34390	0.32596
testing loss	0.45329	0.37592
testing MAE	0.41419	0.40527
testing MSE <sup>†</sup>	0.45329	0.39924
standard deviation of log of $\frac{Z_{pred}}{Z_0}$ <sup>††</sup>	0.34263	0.14875

Table 3: Performance of the two neural networks trained on simulated (171 group) data

<sup>†</sup> MAE = mean-absolute-error; MSE = mean squared error. Since MSE is identically defined as in eq.12, the \* loss value in column 2 is equivalent to \* MSE.

<sup>††</sup> standard deviation of log of  $\frac{Z_{pred}}{Z_0}$  (shortened to std-dev-log(C/E) below) is defined with equation 19





(a) The predicted label (neutron flux as unfolded by the neural network) compared with the original flux. Note that the colour scheme is reversed, i.e. the blue bars denote the reaction rates predicted by the neural network instead of the true reaction rates, and asked to predict the fluence (Figure 6a).

Figure 6: An example of the neutron spectrum as unfolded by the predicting a perturbed JET first wall spectrum.

## 5 Neural networks trained on real spectra

From the result of the previous section, we can expect the neural network to be able to identify the relationship between reaction rates and neutron spectra, while replicating the underlying pattern in the real neutron spectra, when sufficient data is given.

A set of 212 neutron spectra were acquired from the IAEA+UKAEA compendium[37]. 137 of them were identified as fission neutron spectra and 19 of them were identified as fusion neutron spectra. These spectra are listed in the appendices L.

They were rebinned into the Vitamin-J group structure using FISPACT-II[1]. The Vitamin-J group structure was chosen as it is well known and widely used in neutron spectrum unfolding [39] [49] [16]. Using the 11 reactions that are measured by the ACT system [38], the corresponding reaction rates for each spectrum was obtained by folding it through the response matrix (plotted in Figure 7). The method of obtaining the response matrix is explained in Figure 7:

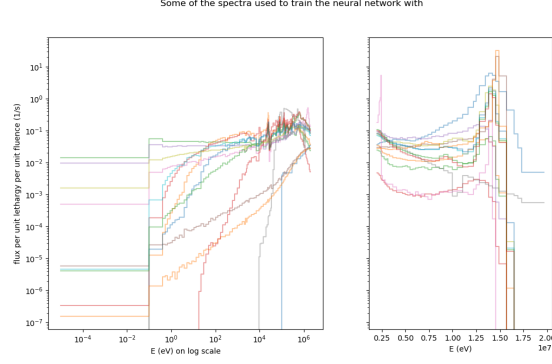


Figure 8: All fusion spectra used, obtained from [37]

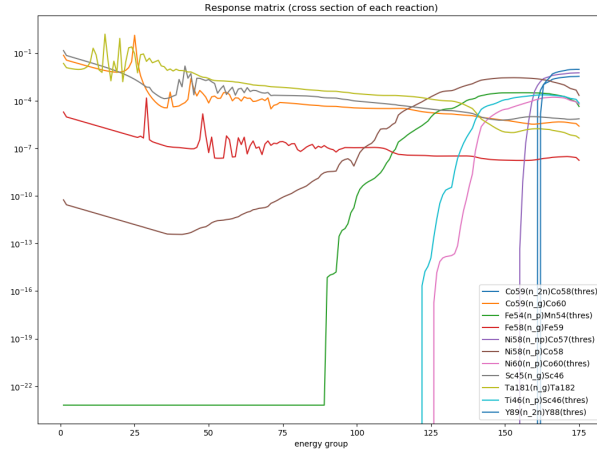


Figure 7: Microscopic cross-section of each reaction

These values are obtained from TENDL15 via FISPACT-II [1] at the left edge of each bin in the Vitamin-J group structure.

By assuming that the flux per unit lethargy inside each bin are relatively flat (energy independent), the reaction rates contributed by the neutron flux in each bin is then proportional to the product of neutron flux with the microscopic. Symbolically,

$$Z_j \propto \sum_i \sigma_{ji} \phi_i \quad (20)$$

Akin to the equation 4. Therefore these microscopic cross-section values were used in place of the response function for each of the reaction, assuming the constant of proportionality in equation 20 is unity.

## 5.1 Hyperparameter Optimization

Since the of each neural network varies according to the hyperparameters used and the data that it is trained on, when investigating the real fusion spectra in section 5, multiple neural networks were generated, each with different hyperparameters, to investigate the combination of optimal hyperparameters which may be applied onto this problem.

The following hyperparameters/variables were considered:

- activation function used
- strategies applied to prevent overfitting

- weight regularization
  - dropout
- number of layers
- number of nodes in each layer
- number of epochs trained
- optimizer algorithm, which has the following parameters:
  - momentum term (if applicable)
  - acceleration term (if applicable)
  - epsilon (denominator offset parameter) (if applicable)
  - learning rate
- Normalization techniques applied:
  - logarithmize the numerical values of features
- metric used to evaluate the loss value (See Section 11)
- Training set/testing set

It is nearly impossible to optimize all 14 parameters listed above simultaneously in a grid search as it will be very computationally intensive and laborious. Therefore the following choices were made for each of the hyperparameter to reduce the amount of variations required for the grid search:

hyperparameter	choice
activation function	ReLU[3] for nodes in all layers, as it is the most widely used activation function in machine learning and simplest function.
overfitting prevention strategies	Early Stopping (stopping the training when the validation loss does not see improvement in 1000 epochs); while the complexity of the model is restricted by limiting the number of layers to 5 or fewer, so neither dropout or weight regularization will be applied.
number of epochs trained	10000 (subject to change by tensorflow's EarlyStopping callback)
<b>number of hidden layers</b>	ranges from 0-5, as this includes all the configurations stated in Table 1.
number of nodes in each layer	with reference to Table 1, the first hidden layer starts with 32 nodes, and logarithmically increases to the last hidden layer, which has 256 nodes. Therefore the six resulting neural network topologies are 11:175, 11:32:175, 11:32:256:175, 11:32:90:256:175, 11:32:64:128:256:175, 11:32:53:90:152:256:175.
Optimizer algorithm	using the Adam optimizer[4] with its default parameters (except for the learning rate, which is specified below), as it is a widely used algorithm in various machine learning projects.
<b>learning rate</b>	ranges from $10^{-2}$ to $10^{-2}$ , logarithmically spaced, 6 steps per decade.
<b>metric used to calculate loss value training set</b>	ranges from equation 12 to 15 Either fusion spectra or fission spectra is used; if fusion spectra is used, then only 80% (15 spectra) are used as the training set (drawn at random), the remaining 20%(4) are reserved for testing.
testing set	fusion spectra
validation set	20% of the testing set, drawn at random

Table 4: Hyperparameters chosen for building neural networks for investigations

For combination of variable hyperparameters in Table 4 (highlighted with bold typeface), a neural network is created; all other hyperparameters are fixed according to the rows in Table 4 with plain font.

## 5.2 Results of predicting using the real spectra

The resulting neural networks were sorted into two groups according to the training set (into "trained on fission" and "trained on fusion"), then each of them were sorted into 4 smaller groups according to the loss value used during training.

For each group of the data, the resulting loss values, MAE, MSE, and std-dev-log(C/E) were all plotted as heatmaps (see Figure 9 for example).

General effects of learning rate and number of layers were identified in this manner. For all of the dataset and loss function used concerned, a general trend of increasing testing loss was identified as the number of layers increases; while the learning rate has

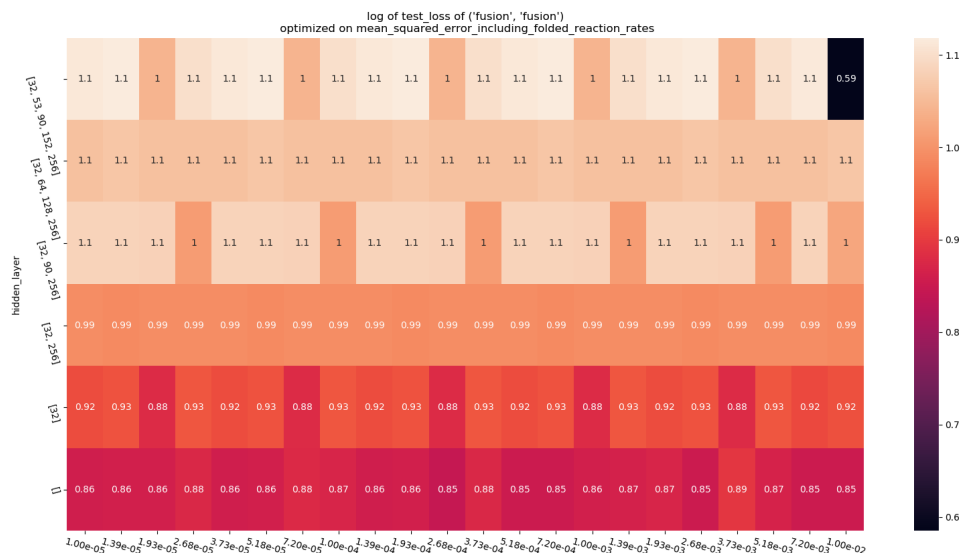


Figure 9: Heatmap visualizing the loss values of the neural networks' prediction on the test dataset.

Each square represents a neural network with a particular set of hyperparameters, i.e. learning rate and number of layers. The learning rate increases logarithmically across the x-axis; while the number of layers increase linearly across the y-axis. The number of nodes per layer is increased logarithmically from 32 to 256 (if the number of hidden layers  $\geq 2$ ). Neural networks which performs better has lower loss values, and are represented with darker colours.

no significant effect on it (except for causing a periodic oscillation in the testing loss as the learning rate increases logarithmically, which was dismissed as an artefact of the Optimizer algorithm Adam).

However, a closer examination of the performance of these few-layer neural networks reveals that their performance are mediocre.

For example, below are some of the predictions made by these few-layer neural networks.

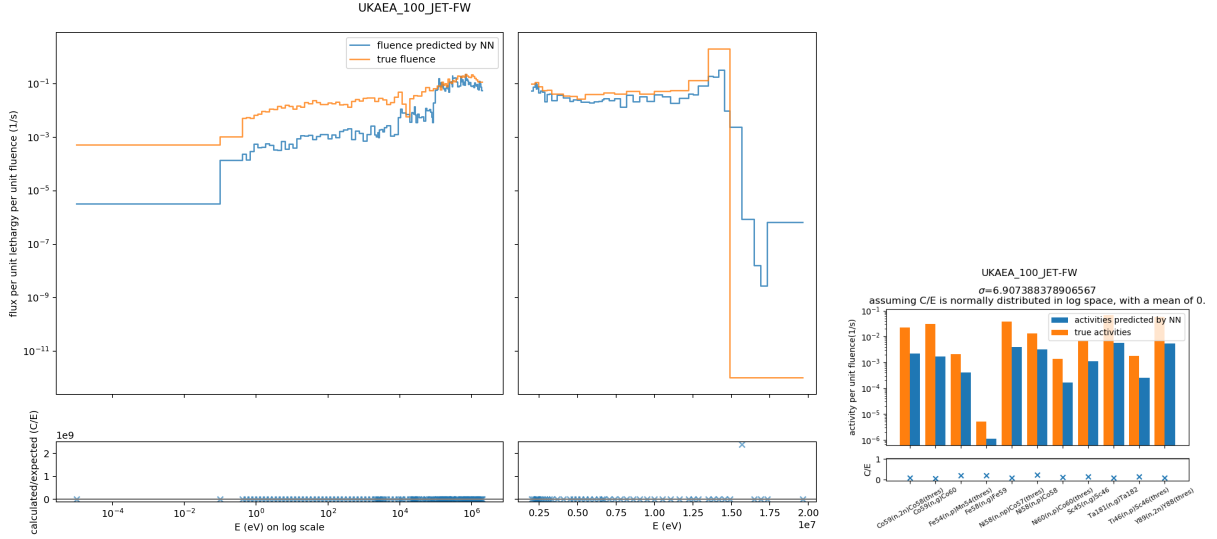


Figure 10: A typical 0-hidden-layer neural network prediction of JET's first wall spectra when trained on fusion spectra (learning rate=0.01)

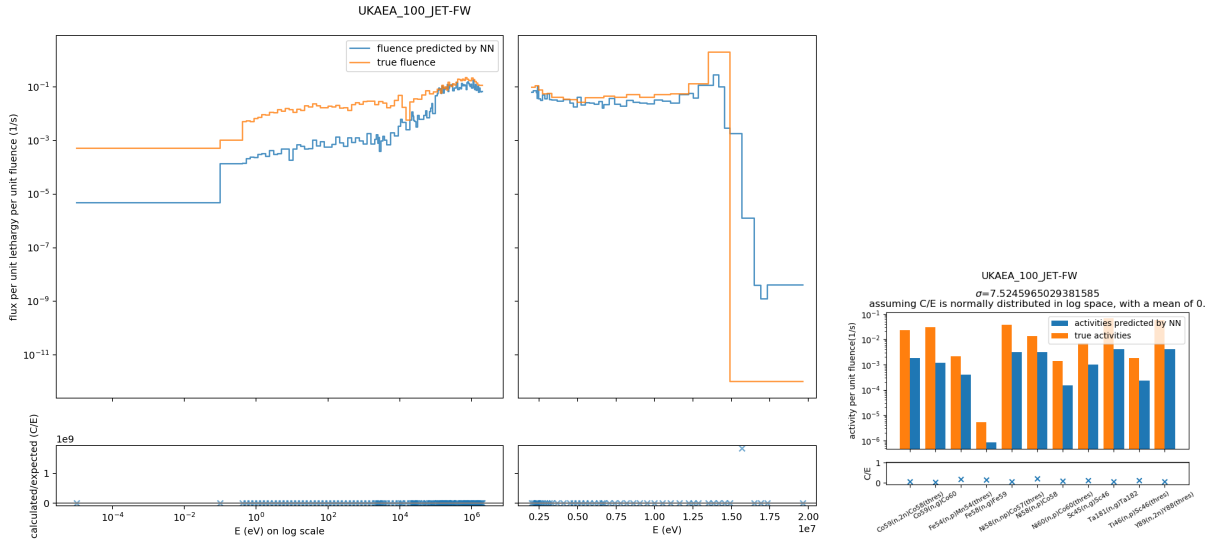


Figure 11: A typical 1-hidden-layer neural network's prediction of JET's first wall spectra when trained on fusion spectra(learning rate=0.01)

They are unable to replicate the 14.1 MeV peak at the correct width, and occasionally massively underestimate the high energy tail.

This is likely due to the small dataset of fusion data available. This explanation is supported by the small number of epochs required for training to finish: most of the neural networks finished training in fewer than 50 epochs, i.e. showed no further improvement in the validation loss in the next 1000 epoch after that. This is unusually early stopping of compared to the number of epochs the neural network required to be trained in Section 4.2.

However, one particular extremum showed up on some of the graphs. This is the neural network with the hyperparameters of (learning rate=0.01, topology=11:32:53:90:152:256:175, loss function used = mean-squared-error-including-folded-reaction-rates, eq. 14), a particularly loss value is obtained. This neural network is found to be as the neural network with the lowest testing loss and testing MSE.

This neural network was able to reproduce the same results even when its initial weights and biases were initialized using a different tensorflow seed. Therefore its low test

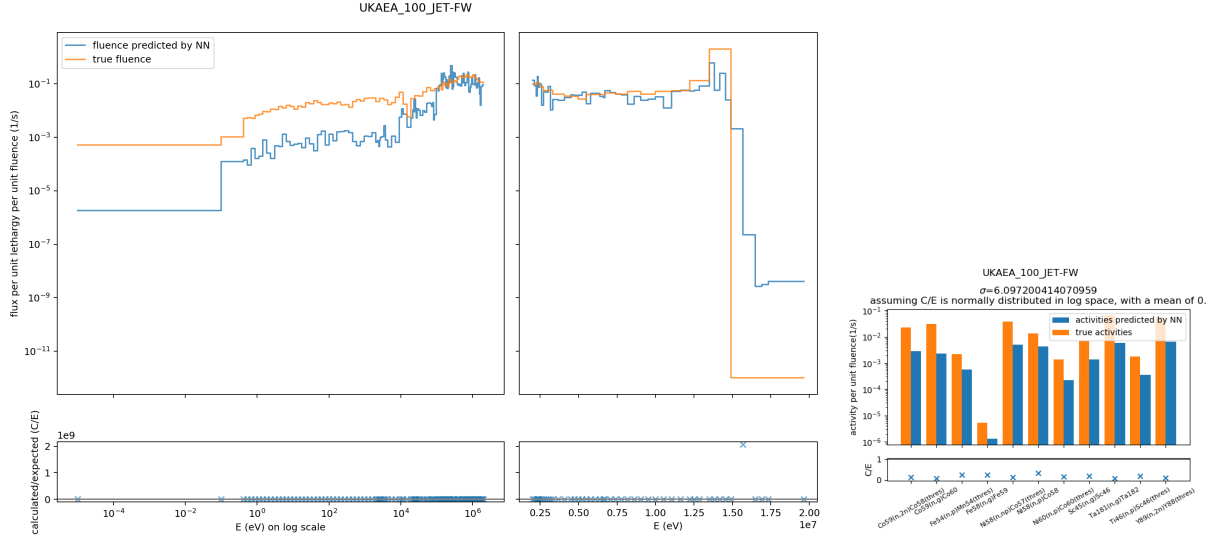


Figure 12: A typical 2-hidden-layer neural network prediction of JET's first wall spectra when trained on fusion spectra (learning rate=0.01)

loss and test MSE were not coincidental, resulting from a numerical instability. It was thought that the neural network was able to generalize from such small dataset, so its predicted spectra were looked into in closer detail.

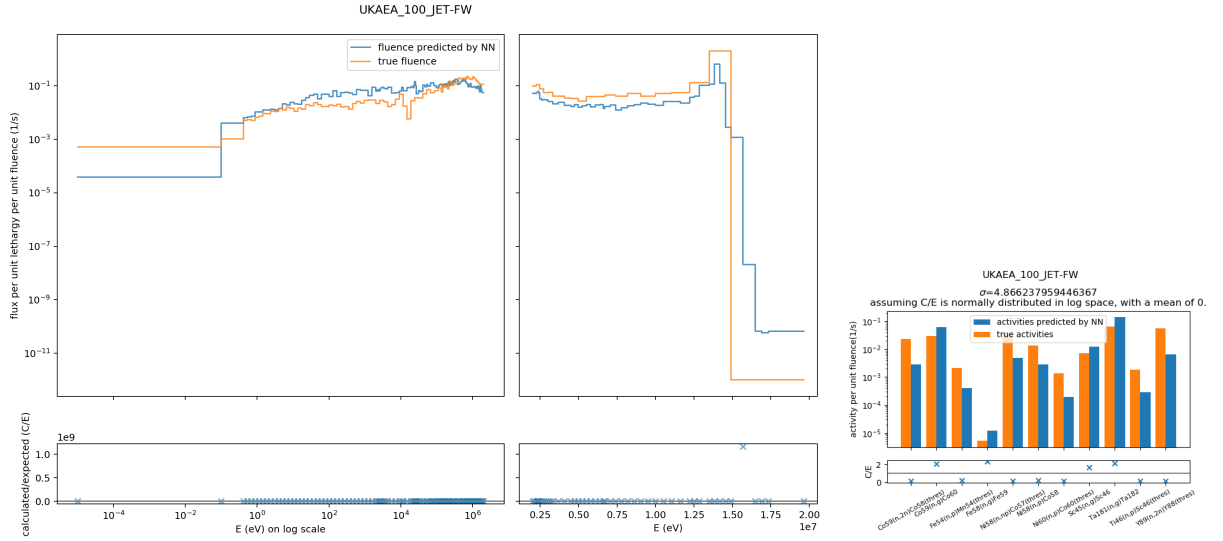


Figure 13: JET first wall spectrum as predicted by the optimally performing NN among all NN trained on fusion data.

But in the end it was concluded that this 5-hidden-layer neural network likely only achieves the above average performance serendipitously by re-tracing the same average spectrum. This hypothesis is supported by it replicating a very similar spectrum when it attempts to deduce the spectra corresponding to the other two test data.

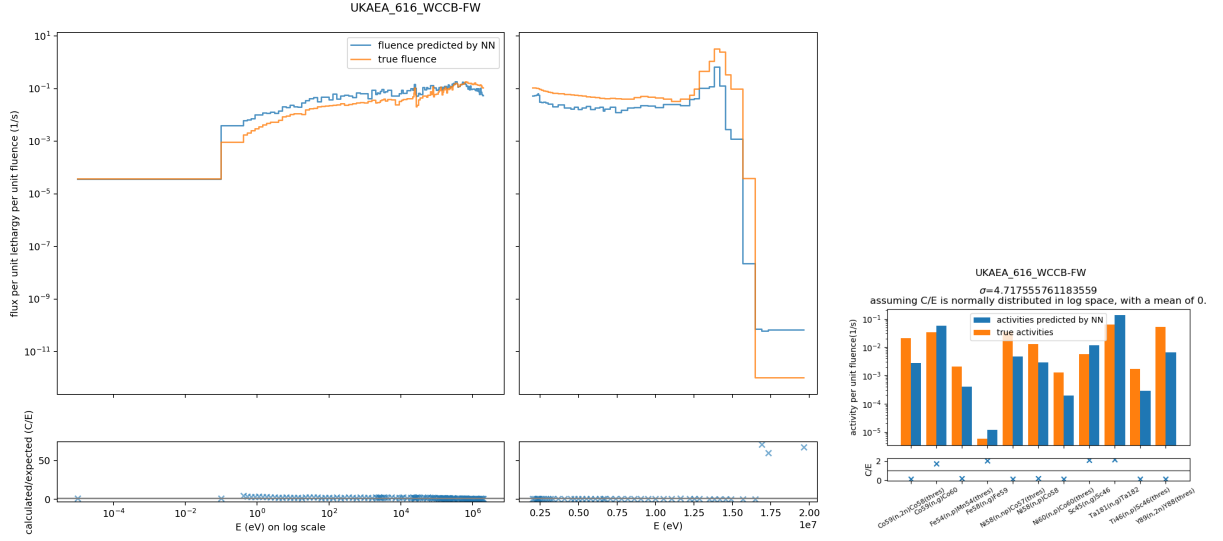


Figure 14: Water-cooled ceramic breeder blanket TOKAMAK's first wall spectrum as predicted by the optimally performing NN among all NN trained on fusion data.

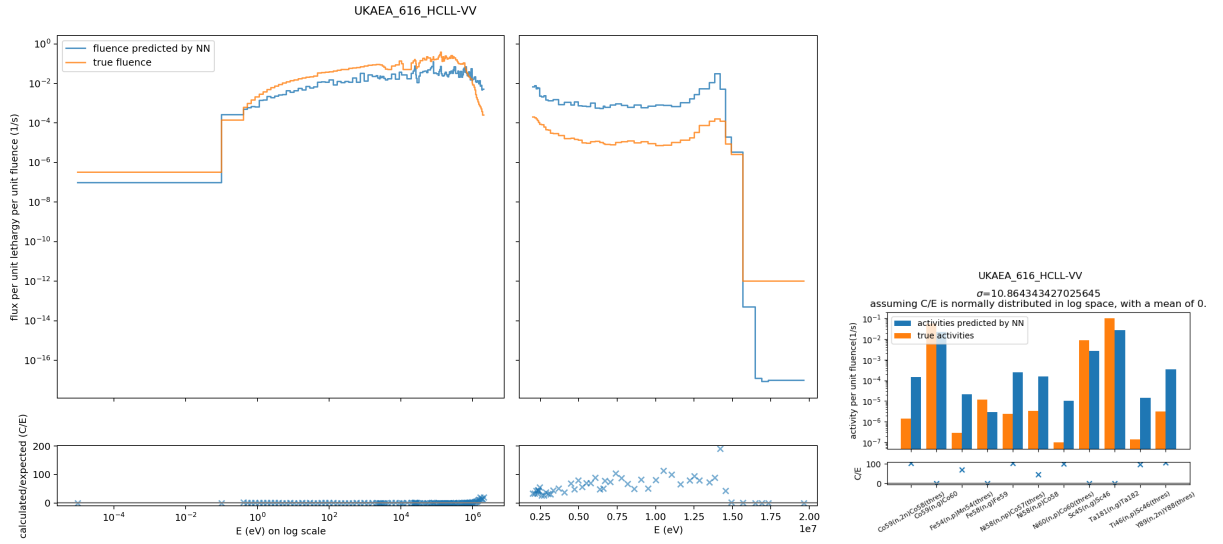


Figure 15: Helium-cooled LiPb TOKAMAK's spectrum at the vacuum vessel, as predicted by the optimally performing NN among all NN trained on fusion data.





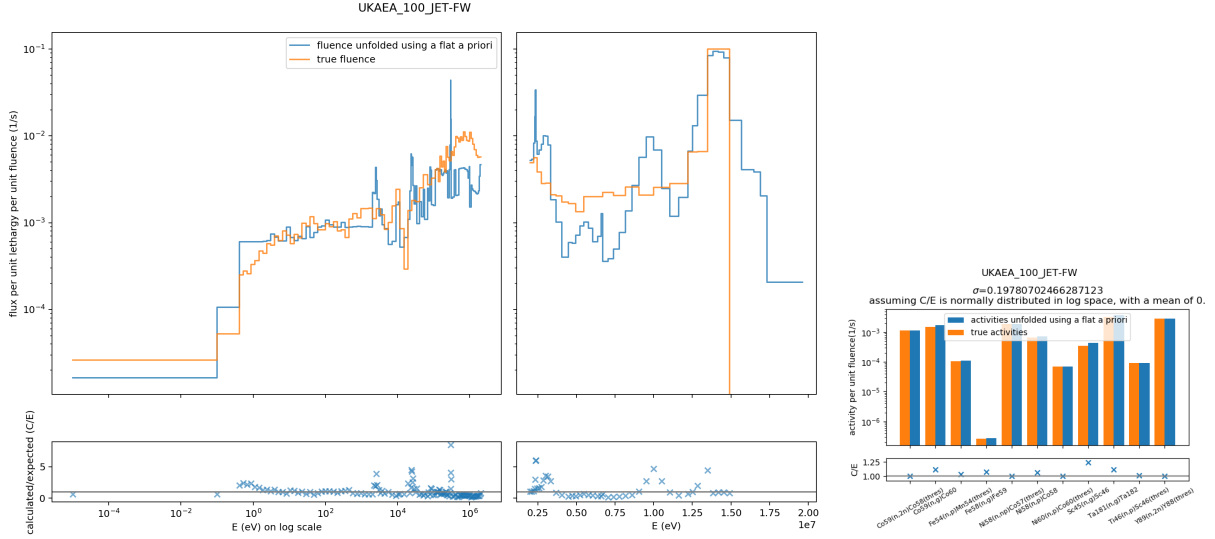


Figure 18: JET first wall spectrum as unfolded by GRAVEL upon using a flat neutron spectrum as the *a priori*

An average MSE value (as defined in equation 12) of 10.188233 was obtained when comparing the unfolded spectra against the 4 true spectra in testing set.

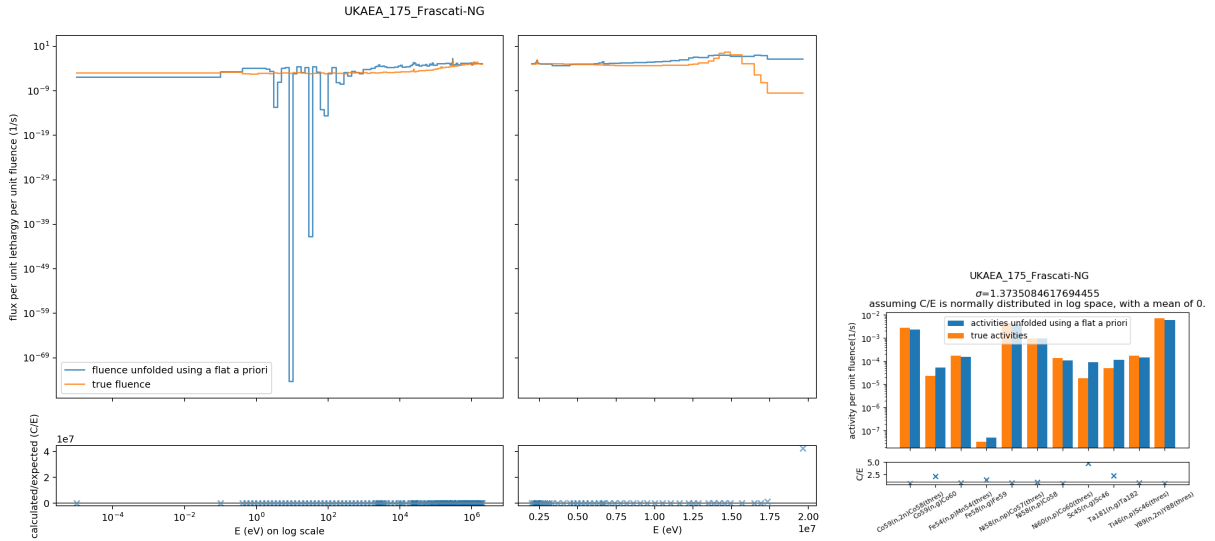


Figure 17: JET first wall spectrum as unfolded by MAXED upon using a flat neutron spectrum as the *a priori*

While MAXED performed very poorly (Figure 17) when given a flat *a priori* (giving an MSE value of 53.442406, as defined in equation 12), GRAVEL was able to unfold a neutron spectrum from the flat *a priori*, though with an underestimated low energy region and overestimated high energy range neutrons. An MSE value of 10.188233 was obtained when comparing the unfolded spectra against the true spectra in log space. (eq. 12)

While the best performing neural network (Figure 13) gives a slightly better result, both Figure 13 and Figure 18 are far from being applicable to real neutron spectrum unfolding, where the unfolding accuracy is much higher (currently the unfolded spectrum gives a total flux that differs from measured total flux by  $\pm 7\%$ [12]). This low accuracy of the solution spectrum is likely due to the small training set of the neural network. This small sample size means that it cannot densely cover the entire domain (feature space) and co-domain (label space) of the fusion neutron spectrum unfolding problem. For the same reason, the neural network's prediction cannot be relied upon for unfolding spectra

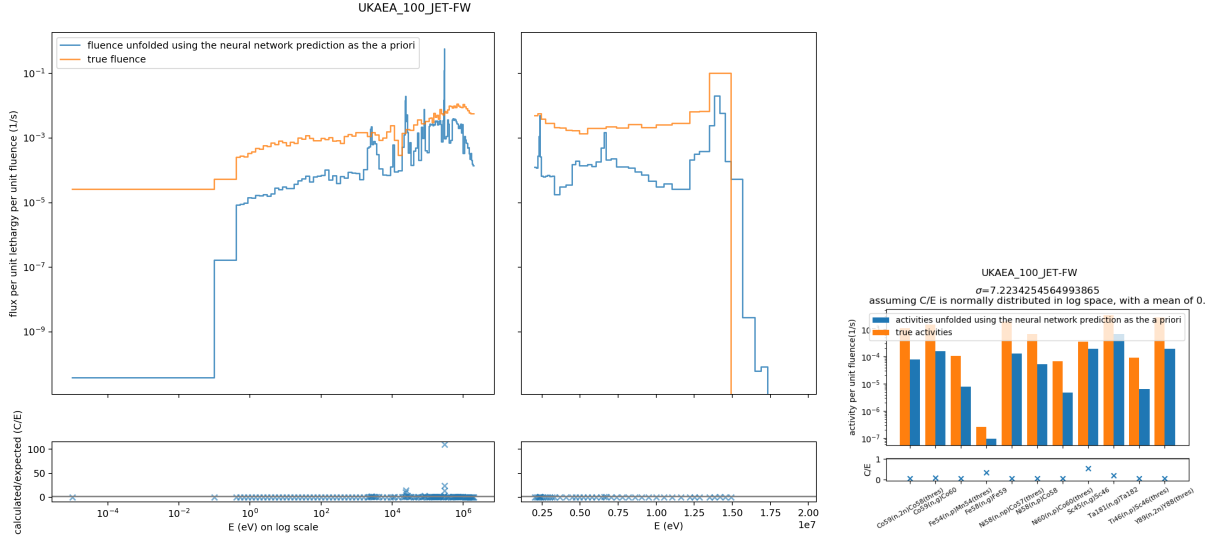


Figure 19: JET first wall spectrum as unfolded by GRAVEL upon using the optimal NN's output as the *a priori* spectrum.

in new/unknown types of fusion reactor designs, as it has not been shown to extrapolate its predictions with satisfactory performance beyond the very short list of fusion spectra in Appendices ??.

### 5.3.2 As an *a priori* generator

When unfolding, an *a priori* spectrum with a reasonably close match to the original spectrum is required as the starting point for an iterative solution search. Given that the neural network did not yield satisfactory unfolding result in the last section, an attempt of using the neural network's output as an *a priori* was made, with the hopes that the neural network's output will be close enough to the true spectrum such that, with further adjustment by the unfolding code such as GRAVEL and MAXED, a reasonable solution spectrum can be reached.

The motivation of using a neural network's output as the *a priori* is that one can save hours of MCNP calculation in order to obtain the response matrix and *a priori* spectrum.

However, this proves to be rather unpromising as well. Even when with best neural network's predictions (Figure 13, 16, 15, 14), GRAVEL unfolded results that are as poor as when a flat *a priori* was used. An average MSE value of 9.89936 was achieved.

And MAXED produced result that are also as meaningless as Figure 17, as the log of the flux in some bins tends to negative infinity.

## 5.4 An attempt at using fission data to predict fusion data

While fusion spectra are scarce, there is an abundant database of fission neutron spectra. Therefore, it would be ideal if neural networks trained on fission spectra can use it to explain fusion spectra, as we will not run into the issue of not having enough data.

Therefore, the results of the second group of neural networks (trained on fission spectra, tested on fusion spectra) were examined. But even the best one of them performed very poorly. It uses an MSE loss value equation 12 giving a testing loss of 16.767817. Closer examination of its prediction shows that it is unsuitable for the neutron spectra.

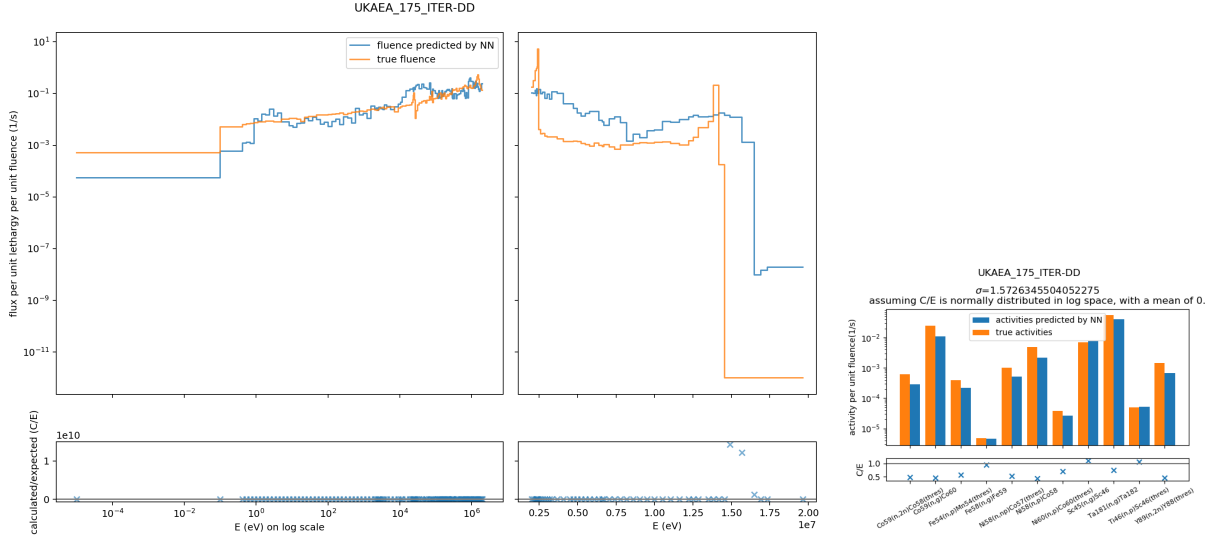


Figure 20: (calculated) ITER spectrum as predicted by the optimal NN among all NN trained on fission spectra

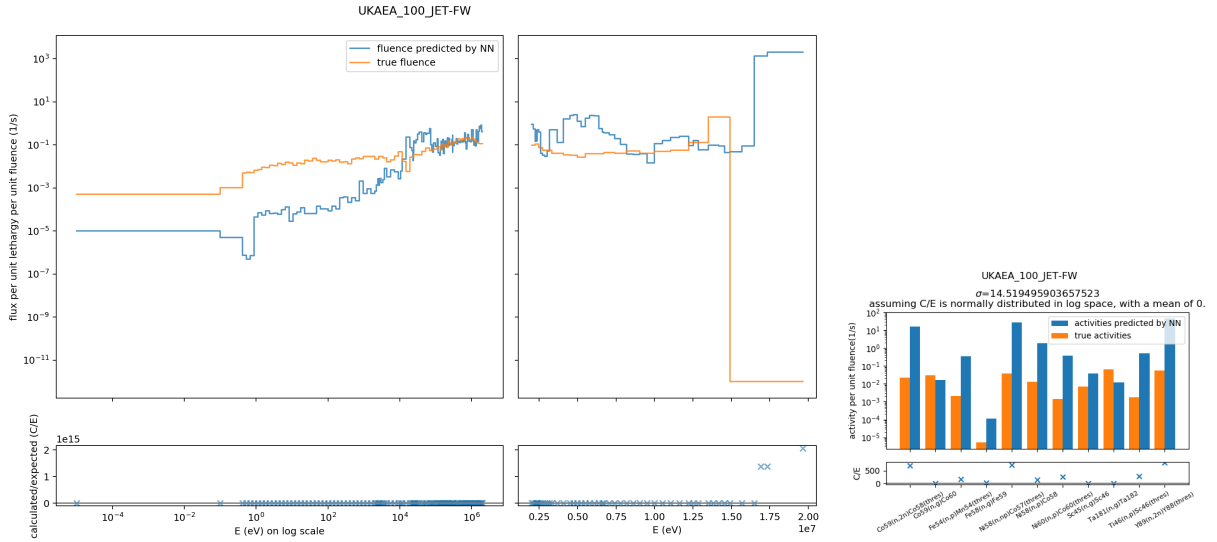


Figure 21: JET first wall spectrum as predicted by the optimal NN among all NN trained on fission spectra

(For the record, this neural network (which performed the best out of all neural networks trained on fission spectra) has very similar hyperparameters as the optimally performing neural network identified in the previous section 5.3.1, differing only in the loss function used. (learning rate = 0.01, topology = 11:32:53:90:152:256:175, loss function used = MSE (eq.12)).)

This suggest that the inverse function required to unfold fission-induced reaction rates in the activation foils is a different inverse function than the inverse function required to unfold the fusion-induced reaction rates in the activation foils.

## 6 Potential Future improvements

The technique of neural network classification is best applied when the dataset is large, which is not the case with fusion neutron spectra.

In order to improve upon the results observed in this thesis, a much larger dataset, consisting the neutron spectra for magnetic confinement fusion with various plasma and first wall conditions, and potentially of other inertial confinement fusion, should be attained. These data may be obtained by numerical simulations with MCNP, or actual measurements of neutron spectra at various TOKAMAK's and various other fusion facilities; but one has to recognize that neural networks trained on a set of data only creates predictions that are as reliable as its training data.

- Use RBF-NN or GRNN, as explained in section 3, which are known to perform better under low sample number conditions.
- Transfer learning[47]: train the neural network with fission data, fix the weights and biases in the half of the neural network, and then train the weights and biases in the remaining half of the neural network using fusion data.
- Use Orthogonal Arrays instead of grid searching the entire hyperparameter space, as well as fractional factorial instead of full factorial combinations, as proposed in [34], when performing the optimization. This can reduce the amount of time required for the experimentation; or extend the range of hyperparameter space searched in the same amount of time; multiple dimensional space can be search through in the same manner as well. This may reveal a combination of hyperparameter which may lead to a better neural network design for fusion neutron spectra unfolding than the existing optimal design.

Lastly, as uncertainty of the neutron spectra needs to be known for the application of fusion neutron measurements, a global sensitivity analysis can be performed on the neural network using a Monte Carlo approach: by repeatedly adding in noise  $\delta Z_j$  into the reaction rates  $Z_j$  and allowing the neural network to carry out its prediction (forward propagation), the Monte Carlo program will be used to infer the resulting changes  $\delta\phi_i$  associated with the neural network's predictions ( $\phi_i$ ), thus propagating uncertainty in the measurements of reaction rates to  $\sigma Z$  to  $\sigma\phi$ . An unfolding code suite is already under development in CCFE to allow this to be done systematically.

## 7 Conclusion

A neural network unfolding approach to neutron spectrum unfolding was examined. A grid search for the optimal hyperparameters for the neural network was performed over 4 dimensions (training set used, learning rate, number of layers and loss function used). The neural network with the best test loss was examined and scrutinized.

A loss value of 10.188 was obtained (MSE in log space, (eq. 12))

- What's the loss values
- achieved using what topology of NN
- trained upon what data
- How does it compare to neutron spectrum unfolding using GRAVEL and MAXED
- Can we use Fission data to train it? (no)
- What's the significance on the unfolding community: should they use it more? Should they improve upon it?
- what additional observation did you find regarding training on different dataset.

## References

- [1] Ukaea: Fistact-ii, 2017.

- [2] explore overfitting and underfitting tensorflow core 2019, 2019.
- [3] tf.nn.relu, 2019.
- [4] tf.train.adamoptimizer, 2019.
- [5] AV Alevra and DJ Thomas. Neutron spectrometry in mixed fields: multisphere spectrometers. *Radiation Protection Dosimetry*, 107(1-3):33–68, 2003.
- [6] Amin Asgharzadeh Alvar, Mohammad Reza Deevband, and Meghdad Ashtiyani. Neutron spectrum unfolding using radial basis function neural networks. *Applied Radiation and Isotopes*, 129:35–41, 2017.
- [7] Matthew Balmer. *Design and testing of a novel neutron survey meter*. PhD thesis, Lancaster University, 2016.
- [8] P Batistoni, D Campling, Sean Conroy, D Croft, T Giegerich, T Huddleston, X Lefebvre, I Lengar, S Lilley, A Peacock, et al. Technological exploitation of deuterium–tritium operations at jet in support of iter design, operation and safety. *Fusion Engineering and Design*, 109:278–285, 2016.
- [9] Cláudia C Braga and Mauro S Dias. Application of neural networks for unfolding neutron spectra measured by means of bonner spheres. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 476(1-2):252–255, 2002.
- [10] FD Brooks and H Klein. Neutron spectrometryhistorical review and present status. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 476(1-2):1–11, 2002.
- [11] Roberto Capote, Konstantin I. Zolotarev, Vladimir G. Pronyaev, and Andrej Trkov. chapter Updating and Extending the IRDF-2002 Dosimetry Library, pages 197–209. ASTM International, West Conshohocken, PA, Aug 2012.
- [12] Bethany Colling, P. Batistoni, S.C. Bradnam, Z. Ghani, M.R. Gilbert, C.R. Nobs, L.W. Packer, M. Pillon, and S. Popovichev. Testing of tritium breeder blanket activation foil spectrometer during jet operations. *Fusion Engineering and Design*, 136:258–264, 2018.
- [13] Fatma Zohra Dehimi, Abdeslam Seghour, and Saddik El Hak Abaidia. Unfolding of neutron energy spectra with fisher regularisation. *IEEE Transactions on Nuclear Science*, 57(2):768–774, 2010.
- [14] Ma. Del Rosario Martinez-Blanco, Gerardo Ornelas-Vargas, Celina Lizeth Castaeda-Miranda, Luis Octavio Sols-Snchez, Rodrigo Castaeda-Miranada, Hector Ren Vega-Carrillo, Jose M Celaya-Padilla, Idalia Garza-Veloz, Margarita Martnez-Fierro, and Jos Manuel Ortiz-Rodrguez. A neutron spectrum unfolding code based on generalized regression artificial neural networks. *Applied Radiation and Isotopes*, 117:8–14, 2016.
- [15] G Fehrenbacher, R Schütz, K Hahn, M Sprunck, E Cordes, JP Biersack, and W Wahl. Proposal of a new method for neutron dosimetry based on spectral information obtained by application of artificial neural networks. *Radiation protection dosimetry*, 83(4):293–301, 1999.
- [16] Lawrence R. Greenwood and Christian D. Johnson. User guide for the staysl pnnl suite of software tools. Technical report, Pacific Northwest National Lab.(PNNL), Richland, WA (United States), 2013.
- [17] Seyed Abolfazl Hosseini. Neutron spectrum unfolding using artificial neural network and modified least square method. *Radiation Physics and Chemistry*, 126:75–84, 2016.
- [18] S Jednorog, E Laszynska, P Batistoni, B Bienkowska, A Cufar, Z Ghani, L Giacomelli, A Klix, S Loreti, K Mikszuta, et al. Activation measurements in support

- of the 14 mev neutron calibration of jet neutron monitors. *Fusion Engineering and Design*, 125:50–56, 2017.
- [19] T. Kin, Y. Sanzen, M. Kamida, K. Aoki, N. Araki, and Y. Watanabe. Artificial neural network for unfolding accelerator-based neutron spectrum by means of multiple-foil activation method. In *2017 IEEE Nuclear Science Symposium and Medical Imaging Conference, NSS/MIC 2017 - Conference Proceedings*. Institute of Electrical and Electronics Engineers Inc., 2018.
  - [20] A Klix, A Domula, U Fischer, D Gehre, P Pereslavitsev, and I Rovni. Neutronics diagnostics for european iter tbms: Activation foil spectrometer for short measurement cycles. *Fusion Engineering and Design*, 87(7-8):1301–1306, 2012.
  - [21] Glenn F. Knoll. *Radiation Detection and Measurement*. Wiley, 2010.
  - [22] Slobodan Krevinac. *Neutron energy spectrum unfolding method*. PhD thesis, University of Birmingham, Birmingham, 1971.
  - [23] C. R. Nobs S. Bradnam B. Colling K. Drozdowicz S. Jednorog M. R Gilbert E. Laszynska D. Leichtle J.W.Mietelski M. Pillon I.E. Stamatelatos T. Vasilopoulou A. Wjcik-Gargula L. W. Packer, P. Batistoni and JET Contributors. Act - activation of real iter materials report on deliverables f21-29. 6 2019.
  - [24] Igor Lengar, Alja Ufar, Vladimir Radulovi, Paola Batistoni, Sergey Popovichev, Lee Packer, Zamir Ghani, Ivan A. Kodeli, Sean Conroy, and Luka Snoj. Activation material selection for multiple foil activation detectors in jet tt campaign. *Fusion Engineering and Design*, 136:988–992, 2018.
  - [25] M. Matzke. Unfolding procedures. *Radiation Protection Dosimetry*, 107(1-3):155–174, 2003.
  - [26] Manfred Matzke. Unfolding of pulse height spectra: the hepro program system. Technical report, SCAN-9501291, 1994.
  - [27] Manfred Matzke. Unfolding methods. *Physikalisch-Technische Bundesanstalt, Germany*, 2003.
  - [28] H Mazrou and F Bezoubiri. Evaluation of a neutron spectrum from bonner spheres measurements using a bayesian parameter estimation combined with the traditional unfolding methods. *Radiation Physics and Chemistry*, 148:33–42, 2018.
  - [29] WN McElroy, S Berg, T Crockett, and RG Hawkins. A computer-automated iterative method for neutron flux spectra determination by foil activation. volume 1. a study of the iterative method. Technical report, ATOMICS INTERNATIONAL CANOGA PARK CA, 1967.
  - [30] Alberto Milocco. Neutron radiation damage in ccd cameras at joint european torus (jet). *Radiation protection dosimetry*, 180(1-4), 2017.
  - [31] Cleve B Moler. *Numerical Computing with MATLAB: Revised Reprint*, volume 87. Siam, 2008.
  - [32] Bhaskar Mukherjee. A high-resolution neutron spectra unfolding method using the genetic algorithm technique. *Nuclear Inst. and Methods in Physics Research, A*, 476(1-2):247–251, 2002.
  - [33] Andrew Ng. Normalizing inputs (c2w1l09), 2017.
  - [34] Jose Manuel Ortiz-Rodriguez, Ma del Rosario Martinez-Blanco, Jose Manuel Cervantes Viramontes, and Hector Rene Vega-Carrillo. Robust design of artificial neural networks methodology in neutron spectrometry. In *Artificial Neural Networks-Architectures and Applications*. IntechOpen, 2013.
  - [35] J.M. Ortiz-Rodrguez, A. Reyes Alfaro, A. Reyes Haro, J.M. Cervantes Viramontes, and H.R. Vega-Carrillo. A neutron spectrum unfolding computer code based on artificial neural networks. *Radiation Physics and Chemistry*, 95:428–431, 2014.

- [36] L. W. Packer, P. Batistoni, S. C. Bradnam, S. Conroy, Z. Ghani, M. R. Gilbert, E. Laszyska, I. Lengar, C.R. Nobs, M. Pillon, S. Popovichev, P. Raj, I.E. Stamatelatos, T. Vasilopoulou, A. Wojcik-Gargula, R. Worrall, and JET contributors. Neutron spectrum and fluence determination at the iter material irradiation stations at jet. 5 2019.
- [37] Lee W. Packer, Mark R. Gilbert, Jonathan Naish, and Steven Bradnam. Ukaea-r-17-nt1 unfolding code support: progress report. 3 2017.
- [38] LW Packer, P Batistoni, SC Bradnam, B Colling, Sean Conroy, Z Ghani, MR Gilbert, S Jednorog, E Łaszyńska, D Leichtle, et al. Activation of iter materials in jet: nuclear characterisation experiments for the long-term irradiation station. *Nuclear Fusion*, 58(9):096013, 2018.
- [39] Prasoon Raj, Steven C. Bradnam, Bethany Colling, Axel Klix, Mitja Majerle, Chantal R. Nobs, Lee W. Packer, Mario Pillon, and Milan tefnik. Evaluation of the spectrum unfolding methodology for neutron activation system of fusion devices. *Fusion Engineering and Design*, 146:1272–1275, 2019.
- [40] M. Reginatto and P. Goldhagen. Maxed, a computer code for maximum entropy deconvolution of multisphere neutron spectrometer data. *Health Phys.*, 77(5):579–83, 1999.
- [41] K Schweda and D Schmidt. Improved response function calculations for scintillation detectors using an extended version of the mcnp code. *Nuclear Inst. and Methods in Physics Research, A*, 476(1-2):155–159, 2002.
- [42] H. Shahabinejad and M. Sohrabpour. A novel neutron energy spectrum unfolding code using particle swarm optimization. *Radiation Physics and Chemistry*, 136, 2017.
- [43] John Shore and Rodney Johnson. Axiomatic derivation of the principle of maximum entropy and the principle of minimum cross-entropy. *IEEE Transactions on information theory*, 26(1):26–37, 1980.
- [44] Everton R Silva, Bruno M Freitas, Denison S Santos, and Cludia L P Mauricio. Algorithm based on artificial bee colony for unfolding of neutron spectra obtained with bonner spheres. *Radiation Protection Dosimetry*, 180(1-4):89–93, 2018.
- [45] Vitisha Suman and P.K. Sarkar. Neutron spectrum unfolding using genetic algorithm in a monte carlo simulation. *Nuclear Inst. and Methods in Physics Research, A*, 737:76–86, 2014.
- [46] D.B. Syme, S. Popovichev, S. Conroy, I. Lengar, and L. Snoj. Fusion yield measurements on jet and their calibration. *Nuclear Engineering and Design*, 246(C):185–190, 2012.
- [47] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A survey on deep transfer learning. *CoRR*, abs/1808.01974, 2018.
- [48] UKAEA. Ukaea: Fistact-ii reference spectra, 2017.
- [49] T. Vasilopoulou, I.E. Stamatelatos, P. Batistoni, N. Fonnesu, R. Villari, J. Naish, S. Popovichev, and B. Obryk. Activation foil measurements at jet in preparation for d-t plasma operation. *Fusion Engineering and Design*, 146:250–255, 2019.
- [50] Jie Wang, Zhirong Guo, Xianglei Chen, and Yulin Zhou. Neutron spectrum unfolding based on generalized regression neural networks for neutron fluence and neutron ambient dose equivalent estimations. *Applied Radiation and Isotopes*, 154, 2019.
- [51] Jie Wang, Yulin Zhou, Zhirong Guo, and Haifeng Liu. Neutron spectrum unfolding using three artificial intelligence optimization methods. *Applied Radiation and Isotopes*, 147:136–143, 2019.



- [52] Ross Worrall. Developing a genetic algorithm for the unfolding of neutron energy spectra. Master's thesis, UNIVERSITY OF BIRMINGHAM, Culham Science Centre, Abingdon OX14 3EB, 9 2014.

# Appendices

## A Neural network building functions tailored for the purpose of neutron spectrum unfolding

The following contains the class in which the neural network is built.

neuralnetworklibrary.py

```
import glob
import sys
import os
# Import commonly used numerical processing and plotting functions
import pandas as pd
import matplotlib as mpl
mpl.use("agg") #for using this script on the cumulus server of ukaea
from matplotlib import pyplot as plt
import numpy as np
from numpy import e
from numpy.fft import fft, ifft
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, activations
import time
from shutil import get_terminal_size
import fcntl

def read_NN_weights(session_name):
    '''returns the weights and biases read from a h5 file'''
    import h5py
    path_to_file = ".checkpoints/" + session_name.split("/")[-1] + ".h5"
    weights, biases = {}, {}
    keys = []
    with h5py.File(path_to_file, 'r') as f: # open file
        f.visit(keys.append) # append all keys to list by visiting each
        for k in keys:
            if ':' in k: # Filter out all keys that is
                # Get the layer number
                name_splitted = f[k].name.split("/")
                layer = name_splitted[1]

                layer_num = "".join(d for d in layer if d.isdigit())
                if layer_num == "": layer_num = "1" # if there is no digit in
                                                    # the layer name: it must'
                                                    # ve been layer 1.

                # Decide whether it's bias or weight according to the last
                                                    # element in the
                                                    # name_splitted list

                if "kernel" in name_splitted[-1]:
                    weights["layer_" + layer_num] = f[k].value
                elif "bias" in name_splitted[-1]:
                    biases["layer_" + layer_num] = f[k].value
    return weights, biases
```

```

def _find_matching_braces(list_of_lines):
    '''given a collection of text lines stored as a list, find out the indices
        of the lines where matching braces
        occurs'''

    # copying the design pattern of finding matching paranthesis.
    brace_stack = [] # stack
    d = {}
    # d stores the opening and closing braces' line numbers
    for l_num, line in enumerate(list_of_lines):
        if "{" in line: brace_stack.append(l_num)
        if "}" in line:
            try:
                d[brace_stack.pop()] = l_num
            except IndexError:
                print("More } than {")
    if len(brace_stack) != 0: print("More { than }")
    return d

def convert_str_value(string):
    if ("[" in string) and ("]" in string): # filter out the list
        splitted_list = string.strip("[]").split(",")
        # filter out the empty list case:
        if len(splitted_list) == 1:
            if splitted_list[0] == "":
                # return an empty list [] instead of a [None]
                return []
        return_list = [convert_str_value(elem.strip()) for elem in
                        splitted_list] # recursively call itself on the
                                      elements of the
                                      list

        return return_list
    if string.startswith('\'') and string.endswith('\''): # filter out the
                                                            strings
        assert string.count('\'') == 2, "too many quotation marks!"
        return string[1:-1]
    if string.startswith('"') and string.endswith('"'): # filter out the
                                                            strings
        assert string.count('"') == 2, "too many quotation marks!"
        return string[1:-1]
    if "False" in string: return False # filter out the booleans and None's
    if "True" in string: return True
    if "None" in string: return None
    if ( "." in string) or ("e-0" in string): # filter out the floats
        try:
            return float(string)
        except ValueError: # filter out the function objects
            if ("<" in string) and (">" in string) and ("object" in string):
                raise ValueError("Cannot input a method object as a string; but
                                   can try using string e.
                                   g. 'AdaGrad'")
    return int(string) # only integers should be left

def cut_file_in_halves(filename):
    '''
    return two lists, one containing the first dictionary;
    the other contains all other files.
    '''
    with open(filename, "r") as f:
        data = f.readlines()

```

```

braces = _find_matching_braces(data)
try:
    first_pair = next(iter(braces.items()))
except StopIteration:
    sys.exit("No more dictionaries in file")
first_dict = []
for line in data[first_pair[0]:first_pair[1] + 1]:
    if ":" in line:
        line = line.strip().strip("{}").strip()
        if line[-1] == ",": line = line[:-1] # remove the rightmost comma
        first_dict.append(line)
rest_of_the_lines = data[first_pair[1] + 1:]
return first_dict, rest_of_the_lines

def convert_lines_to_dict(lines):
    dictionary_to_be_returned = {}
    for line in lines:
        # split the "sentence" down the middle at the ':'
        key, value = [arg.strip() for arg in line.split(":")]
        # must ensure that none of these are empty
        assert not len(key) == 0, "Must have a key before the :"
        assert not len(value) == 0, "Must have a value after the :"
        dictionary_to_be_returned[key] = convert_str_value(value)
    return dictionary_to_be_returned

def overwrite_file_by_removing_first_dict(filename, lines):
    #fcntl.flock(filename, fcntl.LOCK_EX | fcntl.LOCK_NB)
    with open(filename, "w") as f:
        for line in lines:
            f.write(line)
    #fcntl.flock(filename, fcntl.LOCK_UN)

def fold_and_append(response_matrix, label, log_label):
    if log_label: #exponentiate, multiply, and then take log again:
        label_in_linear = tf.math.pow(e, label)
        pred_feature_in_linear = tf.matmul(label_in_linear, response_matrix.T,
                                            astype("float32"))
        pred_feature = tf.math.log(pred_feature_in_linear)
    elif not log_label:
        pred_feature = tf.tensordot(response_matrix, label)
    return tf.concat([label, pred_feature], axis=1)

def convert_str_to_loss_func(string, response_matrix, log_label):
    include_folding_string = "_including_folded_reaction_rates"
    if string.endswith(include_folding_string):
        loss_func = convert_str_to_loss_func( string.replace(
                                                    include_folding_string, ""),
                                                    response_matrix, log_label=
                                                    log_label) #use to get one of
                                                    the following
        return lambda lab, pred : loss_func( fold_and_append(response_matrix,
                                                                lab, log_label=log_label),
                                                                fold_and_append(response_matrix,
                                                                pred, log_label=log_label)) #
                                                                return a wrapped function
        #This assumes that each element of the folded reaction rate has the
        same weight in terms of
        deviation from the label.
    elif string=="mean_squared_error":
        return tf.compat.v1.losses.mean_squared_error

```

```

elif string=="mean_pairwise_squared_error":
    return tf.compat.v1.losses.mean_pairwise_squared_error
elif string=="cosine_distance":
    return lambda lab,pred : tf.losses.cosine_distance(lab,pred,axis=0)
else:
    return string

```

```

class NeuralNetwork():
    #This class contains all the read- and write information required to pre-
    #process and post-process inputs to
    #the neural network.

    #It allows for all types of input imaginable, except for the
    def __init__(self):
        # List of parameters for pre- and post-processing
        self.data_preparation_options = {
            "log_feature"      : True,
            "log_label"       : True,
            "lower_limit"     : 1E-12, # any flux value below lower_limit will
                                     # be clipped to
                                     # lower_limit

            "label_already_in_PUL" : False, #labels needs to be converted
            #from total flux per bin to average flux per unit lethargy (PUL
            #) across the bin before
            #training/handled by the
            #NN.

            #total flux needs to be divided by the difference in lethargy
            #of the upper and lower
            #limit to be converted
            #into flux PUL.

            "ft_label"        : False, # Do not apply fourier transform
            #before processing the
            #data by default.

            # apply log on both sides (the RR and the flux) before
            #processing the data, by
            #default
        }

        # options of how to rearrange the data before reading it in.
        self.data_reordering_options = {
            "shuffle_seed"      : 0,
            "startoff"          : None,
            "cutoff"             : None, #number of data lines to accept
            #from the next file.

            "train_split"       : 0.8,
            "validation_split"  : 0.2,
        }

        # metadata recording the training time. These will be auto-generated as
        #the NeuralNetwork training
        #begins.

        self.timing = {
            "start_time_raw"    : time.time(), #give in unix time
            "start_time"        : time.strftime("%I:%M%p %d-%m-%Y").lower(),
            #
            "run_time_seconds"   : 0.0,
        }

        start_time_global = self.timing["start_time_raw"]

```

```

# hyperparameter describing the architecture of the NN
self.hyperparameter = {
    "tf_seed"      : 0,
    "act_func"     : [],
    "hidden_layer" : [],
    "learning_rate" : 0.001,
    "loss_func"    :
        "mean_pairwise_squared_error",
        # "cosine_distance", # chi^2 calculated as
                                normalized
                                unit sum of
                                squared
                                values #this
                                one is
                                weird and I
                                can never
                                get it to
                                work.
        # "mean_squared_error", #chi^2 calculated as mean
                                of squares
                                of deviation
                                from true
                                labels.

    "metrics"      : ['mean_absolute_error', 'mean_squared_error'], #
                                "precision_at_thresholds"
                                only works with boolean,
                                therefore is not used.

    "num_epochs"   : 10000,
}
self.hyperparameter["optimizer"] = tf.keras.optimizers.Adam(self.
    hyperparameter["learning_rate"])

#loss values to be filled in later
self.losses = {
}
# for key in self.hyperparameter['metrics']:
#     self.losses.update({key: 0})

self.session_name = ""

self.callbacks_applied = ["PrintEpochInfo"]#, "TensorBoard"]

# list the parameters to be saved
self.settable_property_list = list(self.__dict__.keys())

##### Everything above
                                may be tweaked manually before
                                starting building and training;
##### Everything below
                                will be automatically generated
                                and shared across the class.

# class instances of callbacks; used for monitoring training in real
                                time/reviewing it afterwards..
class _PrintEpochInfo(keras.callbacks.Callback): # inherits from keras
    .callbacks.Callback,
    # which is a dummy class specifically designed for creating objects
                                that goes into callbacks
                                argument in tf.model.fit();
    # This is a local class that will not need to be reused outside of

```

```

                                the function.
start_time_global = self.timing["start_time_raw"]#grab the global
                                start time from the timing
                                dictionary above.
def on_epoch_end(self, epoch, logs): # redefine the function so
                                that it prints only a dot,
                                regardless of verbosity
                                level.

    # ignore the logs (which logs the mae and mse)
    terminal_width = get_terminal_size().columns

    output_string = "{:>7} epochs finished;"\
                    "loss-value (mse) = {:.9f};"\
    "validation loss-value (mse) = {:.9f};"\
    "program has ran for = {:.04.2f} s".format(
        epoch + 1, logs["loss"], logs["val_loss"], time.time()
                                -
                                start_time_global
                                )

    prompt_wider = "please make the terminal wider!"
    if terminal_width>=len(output_string):
        print( output_string ,end="\r", flush=True) # make sure
                                                the screen is wide
                                                enough to print all
                                                of this in a single
                                                line; otherwise it
                                                will overflow into
                                                the next line then
                                                the "\r" and flush
                                                operation will not
                                                extend back onto the
                                                first line, and the
                                                flush behaviour won
                                                't occur.

    elif len(prompt_wider)<=terminal_width<len(output_string):
        print( prompt_wider, end="\r", flush=True)
    else:
        pass #don't print anything.

if not os.path.exists(".checkpoints/tb_logs/"): os.makedirs(".
                                checkpoints/tb_logs/")
self.callback_objects_available = {
    "PrintEpochInfo" : _PrintEpochInfo(), #just to print the epoch info
                                                to screen.
    "TensorBoard" : tf.keras.callbacks.TensorBoard(log_dir=".
                                                checkpoints/tb_logs/
                                                latest_run", histogram_freq=
                                                1), #overwrites the
                                                previously saved TensorBoard
                                                file.
    "EarlyStopping" : tf.keras.callbacks.EarlyStopping(patience=1000,
                                                restore_best_weights=True),
    "ProgbarLogger" : tf.keras.callbacks.ProgbarLogger(),
    "ReduceLROnPlateau": tf.keras.callbacks.ReduceLROnPlateau(),
}

self.keep_showing_figure = True #this has to be kept in order to make
                                things simple and modular.

```

```

self.folder = "test"

#recording the model itself
self.model = None

# A list of variables used for sharing numerical data/object across
                                methods.
self.data_input = {
    # This dictionary only stores the corresponding data,
    # all of which are stored in the format of DataFrame
    "feature_before_preprocessing" : None,
    "train_feature" : None,
    "test_feature" : None,

    "label_before_preprocessing" : None,
    "train_label" : None,
    "test_label" : None,

    "true_spec" : None, # in usual operation, post-processing "
                                test_label" will give "
                                true_spec";
                                # i.e. the testing split of the trimmed "
                                label_before_prep
                                " is
                                identical
                                to
                                true_spec
                                .

    "ref_spec" : None, # a THIRD line to be plotted on the graph. This
                                is only utilized when
                                predicting the demo data.

    "ref_info" : None, # dataframe from which the title text is loaded
                                .

    "group_structure" : None,
    "response_matrix" : None,
}

self.evaluation_output = {
    # this is a hybrid dictionary that stores data in various formats (
                                numpy.array, pandas.
                                DataFrame, list).

    "hist_df" : None,
    "predicted_labels_array_before_post_processing": None, # Holds the
                                prediction values (from file
                                or from test set)

    "predicted_labels_array_after_post_processing" : None,
    "error" : [], # list of elementwise error
}

def interactive_neural_network_maker(self):
    key_input_prompt = "input the any key or attribute whose value that you
                                'd like to change, or input 'c'
                                to exit:"

    for d in self.settable_property_list:
        print("{0} :".format(d))
        print(getattr(self,d), "\n")
    while True:
        key_input = input(key_input_prompt)
        if key_input=="c":

```

```

        break
    for d in self.settable_property_list:
        val_input_prompt = "input the value for {0} as you would in
                             python script ('quotes'
                             around str, [brac]
                             around lists, etc.):".
                             format(d)

        if type(getattr(self,d))==dict:
            keys = getattr(self,d).keys()
            for k in keys:
                if key_input.strip()==k:
                    val_input = convert_str_value(input(
                                                                val_input_prompt
                                                                ))

                    dict_copy = getattr(self,d)
                    dict_copy[k]=val_input
                    setattr(self,d,dict_copy)
                    print(d, "now takes the value of ", dict_copy)
            elif key_input.strip()==d:
                val_input = convert_str_value(input(val_input_prompt))
                setattr(self,d,val_input)
                print(d, "now takes the value of ", val_input)

def try_to_update_attribute(self, test_k, value):
    if hasattr(self, test_k):
        setattr(self, test_k, value)
        return
    else:
        dictionaries = [ i for i in dir(self) if type( getattr(self,i) )==
                           dict] #get the list of
                                   attributes which are
                                   dictianaries.

        for dic_name in dictionaries:
            if test_k in getattr(self,dic_name).keys(): # if the input key
                                                         is found in the
                                                         dictionary.

                dic_copy = getattr(self, dic_name) #get a copy of the
                                                         dictionary

                dic_copy[test_k] = value # change the corresponding value
                setattr(self, dic_name, dic_copy)
                return # only stop retun the method if we stop the case.
            raise KeyError("no attribute or key named", test_k)

def load_data(self, csv_file, data_input_key):
    """
    Retrieve data from .csv in the same directory without normalziation;
    Usual use case is
    nn.load_data("reaction_rate.csv","feature_before_preprocessing")
    nn.load_data("flux.csv", "labels_before_preprocessing")
    """
    df = pd.read_csv(csv_file, delimiter="," , header=None, comment="#")

    # Error-checking:
    # Ensure that the data obtained are of the correct size before saving
    # it as a class attribute.

    if "label" in data_input_key:
        opposite_key = data_input_key.replace("label", "feature")
    elif "feature" in data_input_key:
        opposite_key = data_input_key.replace("feature", "label")
    elif data_input_key=="ref_spec":

```



```

        opposite_key = "feature_before_preprocessing"
    elif data_input_key=="true_spec":
        opposite_key = "test_label"
    elif data_input_key=="group_structure":
        pass #ignore this case
    elif data_input_key=="response_matrix":
        df = pd.read_csv(csv_file, header=None, index_col=0) #redo the read
                                                             , including the indices name
                                                             for each row.

    elif data_input_key=="ref_info":
        df = pd.read_csv(csv_file, header="infer") #redo the read,
                                                    including the column headers
        opposite_key="label_before_preprocessing"
    else:
        raise KeyError( "data_input_key='{0}' not found".format(
            data_input_key) )

    #by asserting that the opposite entry is of the same shape if it has
    #been loaded:
    if data_input_key=="group_structure": #specific treatment for loading
                                         group_structure.

        num_boundaries = len(df.values.flatten())
        assert num_boundaries == max( np.shape(df) ), "The .csv where the
            group_structure is stored" \
            "must contain only a single line of data, stored vertically or
            horizontally"

    if type(self.data_input["label_before_preprocessing"])!=type(None):
        label_num_col = len(self.data_input["label_before_preprocessing"]
            .columns)
    elif type(self.data_input["train_label"])!=type(None):
        label_num_col = len(self.data_input["train_label"].columns)
    try:
        assert num_boundaries==( label_num_col+1 ), "there must be N+1
            "\
            "group boundaries value provided for N flux values provided
            ; "\
            "But at the moment the group_structure has length = {1} "\
            "which doesn't match the second dimension of train_label's
            boundary "\
            "{0}".format( num_boundaries , np.shape(self.data_input["
            train_label"] ) )
        #Check that the shape of group_structure corresponds with
        the labels.
    except UnboundLocalError as E:
        if "label_num_col" in str(E):
            pass # this means the group structure was loaded before "
                train_label" or "
                label_before_preprocessing
                "

    elif data_input_key == "response_matrix":
        index_len, columns_len = df.shape
        if type(self.data_input["label_before_preprocessing"])!=type(None):
            label_col_len = len(self.data_input["label_before_preprocessing"]
                .columns)
            assert label_col_len==columns_len, "number of columns in the
                response matrix({1})
                must equal to the number
                of neutron groups({0})"
            .format(label_col_len,

```

```

        columns_len)
    if type(self.data_input["feature_before_preprocessing"]) != type(None):
        feature_col_len = len(self.data_input["feature_before_preprocessing"].columns)
        assert feature_col_len == index_len, "number of activities in features({0}) must equal to the number of rows in the response matrix({1}).".format(feature_col_len, index_len)
    elif type(self.data_input[opposite_key]) != type(None):
        assert len(self.data_input[opposite_key].index) == len(df.index), "The entries in {0} must have one-to-one correspondance" \
            "with the entries in {1}. But they have shape {2} and {3}" \
            "respectively".format(data_input_key, opposite_key, np.shape(df), np.shape(self.data_input[opposite_key]))
    assert not (df.isnull().values.any()), "NaN value(s) found inside dataframe!"

    #saving the dataframe as an attribute to be used across the class.
    self.data_input.update({data_input_key:df})

def _preprocess_numerical_values(self, df_or_array, datatype):
    assert (datatype=="feature") or (datatype=="label"), "The datatype must be specified either as 'label' or 'feature'."

    if datatype=="feature":
        if self.data_preparation_options["log_feature"]:
            df_or_array = np.log(df_or_array)
            df_or_array = np.clip(df_or_array, np.log( self.data_preparation_options["lower_limit"] ), None)
            #
    if datatype=="label":
        if not self.data_preparation_options["label_already_in_PUL"]:
            df_or_array = self._convert_to_PUL(df_or_array)
        if self.data_preparation_options["log_label"]:
            df_or_array = np.log(df_or_array)
            df_or_array = np.clip(df_or_array, np.log( self.data_preparation_options["lower_limit"] ), None)
            #clip all values to above zero to prevent -inf's when taking log.
        if self.data_preparation_options["ft_label"]:
            df_or_array = fft(df_or_array)
    return df_or_array

def trim_data(self): # self.data_reordering_options["cutoff"]
    """
    Cut out unused data from self.data_input["feature"] and self.data_input["label"]
    using self.data_reordering_options["cutoff"]
    """
    cutoff_point = self.data_reordering_options["cutoff"] #copying the global cutoff variable to a

```

```

                                shorter expression.
startoff_point = self.data_reordering_options["startoff"]
if (cutoff_point==None) and (startoff_point==None): print("trim_data
                                called but data is not trimmed
                                since startoff and cutoff=None")
self.data_input["feature_before_preprocessing"] = self.data_input["
                                feature_before_preprocessing"][
                                startoff_point:cutoff_point]
self.data_input["label_before_preprocessing"] = self.data_input["
                                label_before_preprocessing"][
                                startoff_point:cutoff_point]
if type(self.data_input["ref_spec"])!=type(None): #if ref_spec is not
                                empty:
    self.data_input["ref_spec"] = self.data_input["ref_spec"][
                                startoff_point:cutoff_point]
if type(self.data_input["ref_info"])!=type(None):
    self.data_input["ref_info"] = self.data_input["ref_info"][
                                startoff_point:cutoff_point]

def shuffle(self): # self.data_reordering_options["shuffle_seed"]
    """
    shuffle the *_before_preprocessing DataFrames in self.data_input to a
                                random but reproducible order
    using self.data_reordering_options["shuffle_seed"]
    """
    assert len(self.data_input["feature_before_preprocessing"])==len(
        self.data_input["label_before_preprocessing"]), "features and
                                labels must have 1-to-1
                                correspondance."
    indices = np.arange(len(self.data_input["feature_before_preprocessing"]
                                ))
    if self.data_reordering_options["shuffle_seed"] != None:
        np.random.seed(self.data_reordering_options["shuffle_seed"])
        np.random.shuffle(indices) # operate in-place
    else:
        print("shuffle is called but data is not shuffled since
                                shuffle_seed=None")
    self.data_input["feature_before_preprocessing"] = self.data_input["
                                feature_before_preprocessing"].
                                loc[indices]
    self.data_input["label_before_preprocessing"] = self.data_input["
                                label_before_preprocessing"].loc
                                [indices]
    if type(self.data_input["ref_spec"]) != type(None):
        self.data_input["ref_spec"] = self.data_input["ref_spec"].loc[
                                indices]
    if type(self.data_input["ref_info"]) != type(None):
        self.data_input["ref_info"] = self.data_input["ref_info"].loc[
                                indices]

def split_into_sets(self): # self.data_reordering_options["train_split"]
    """
    populate train_* and test_*
    by splitting *_before_preprocessing in two parts
    according to the fraction determined by self.data_reordering_options["
                                train_split"]
    """
    print("populating sets from *_before_preprocessing...")
    sample_size = len(self.data_input["feature_before_preprocessing"]).index
                                )

```

```

# Use the first part as training data, the second part as
num_train = round(self.data_reordering_options["train_split"] *
                  sample_size)

self.data_input["train_feature"] = self.data_input["
    feature_before_preprocessing"].
    iloc[:num_train]
self.data_input["test_feature"] = self.data_input["
    feature_before_preprocessing"].
    iloc[num_train:]

self.data_input["train_label"] = self.data_input["
    label_before_preprocessing"].
    iloc[:num_train]
self.data_input["test_label"] = self.data_input["
    label_before_preprocessing"].
    iloc[num_train:]

if type(self.data_input["ref_spec"])!=type(None):
    self.data_input["ref_spec"] = self.data_input["ref_spec"][num_train
:]
if type(self.data_input["ref_info"])!=type(None):
    self.data_input["ref_info"] = self.data_input["ref_info"][num_train
:]

def preprocess_input(self): # self.data_preparation_options
    ,,,
    Transform the numerical values inside the dataframe (in self.data_input
    ) (reversibly)
    using the options listed in self.data_preparation_options
    ,,,
    #pick out ONLY the test_* and train_* labels and features; leaving the
    *_before_preprocessing alone.
    df_list = [ df_key for df_key in self.data_input.keys() if ("_feature"
    in df_key) or ("_label" in
    df_key) ]

    for k,v in self.data_input.items():
        if type(v) != type(None): # filter out all empty cases
            if "feature" in k:
                v = self._preprocess_numerical_values( v , "feature")
            if "label" in k:
                v = self._preprocess_numerical_values( v , "label")
            self.data_input.update({k:v})

def _print_module_name(self): # dependent on whether build_model is called
    with print_pretty_logo=True or False
    .
    print("'|. '|| ")
    print(" |'| | .... .. .. .. .. || ")
    print(" | '|. | .|...|| || || ||' ' '|| || ")
    print(" | ||| || || || || .'| || || ")
    print(" .|. '| '||... '||..|. .||. '||..'|' .||. ")
    print(" ")
    print(" ")
    print("'|. '|| ")
    print(" |'| | .... .||. .... .. .. .. .. || .. ")
    print(" | '|. | .|...|| || || || | .'| .| ||' ' ||. ")
    print(" | ||| || || || || || || || ||'|. ")
    print(" .|. '| '||... '||. | | '||..|. .||. ||. ")

def build_model(self, print_pretty_logo=True): # self.hyperparameter

```

```

'''
using arguments saved in self.hyperparameter
(which includes tf_seed ,hidden_layer ,act_func ,learning_rate ,
                    loss_func ,metrics)
a model is generated.
'''
tf.random.set_random_seed(self.hyperparameter["tf_seed"])

# act_func should be a list
act_func_iter = iter(
    [activations.linear, ] + self.hyperparameter["act_func"]) # ensure
                                                                that the first layer is a
                                                                purelin activation

def get_next_activation_function(): # create a short method to iterate
                                    through activation functions
    try:
        act = next(act_func_iter)
    except StopIteration:
        act = activations.relu # if user hasn't given enough
                                activation functions,
                                pad the rest using relu.

    return act

neural_network_structure = []
for n in self.hyperparameter["hidden_layer"]:
    if type(n) == int: # if it is an integer, interpret it as "numebr
                        of nodes to insert into the
                        next layer",
        neural_network_structure.append(layers.Dense(n, activation=
                                                    get_next_activation_function
                                                    ())) # and match it to
                                                    the next activation
                                                    function on the list.

    elif type(n) == float:
        assert 0 < n < 1, "a float value is interpreted as a drop out
                            rate, thus must be a
                            fraction between 0 and 1
                            ."
        neural_network_structure.append(layers.Dropout(n))

# The zeroth and last layer have linear activation functions
# and shape corresponding to the input and output respectively.
neural_network_structure.append(layers.Dense(len(self.data_input["
train_label"].columns),
                                              activation=activations.linear))

# first_layer_size = first integer value, otherwise if there are no
hidden layers, then it equals
the number of labels
first_layer_size = len(self.data_input["train_label"].columns)
for n in self.hyperparameter["hidden_layer"]:
    if type(n) == int:
        first_layer_size = n
        break
# forcefully overwrite the first layer to have a purelin activation
function,
# and make sure the zeroth layer understands the input shape to be of
shape=self.num_feature
neural_network_structure[0] = layers.Dense(first_layer_size,
                                              input_shape=[len(self.data_input

```

```

        ["train_feature"].columns)],
        activation=activations.

#getting the loss function:
loss_func = convert_str_to_loss_func(self.hyperparameter["loss_func"],
                                     self.data_input["response_matrix"], self.
                                     data_preparation_options["log_label"])

model = keras.Sequential(neural_network_structure)
model.compile(
    # loss="mean_squared_error",
    # loss="logcosh",
    loss=loss_func,
    # Mean squared error is the most sensible and widely chosen option
    # among all loss functions in this case,
    # where where we're performing a regression with no other boundary
    # condition (e.g. area under graph =1) applied.
    # But perhaps later we may wish to define some functions to
    # penalize for discontinuity between bins,
    # e.g.
    # def loss(x): return abs(np.diff(x))
    optimizer=self.hyperparameter["optimizer"], # use the RMS
                                                    propagation algorithm listed
                                                    above
    metrics=self.hyperparameter["metrics"] #*****Look at changing the
                                                    loss function and metrics!!!
    # save these parameters into the history object such that the
    # accuracy of the NN to the validation set can be tracked.
)
if print_pretty_logo:
    self._print_module_name()
# save these parameters as the class attributes
self.optimizer = model.optimizer # save the optimizer
self.model = model

def _print_params_as_dictionary(self): # dependent on whether train_model
                                     is called with
                                     print_dict_before_training=True or
                                     False.
    '''print all non-numerical parameters and hyperparameters to stdout'''
    dictionary_of_params = {}
    for k in self.settable_property_list:
        dictionary_of_params[k] = getattr(self, k)
    for k, v in dictionary_of_params.items():
        print(k, ":", v, "\n")

def train_model(self, print_dict_before_training = True, verbose=0): # "
                                     num_epochs", "validation_split",
                                     callbacks_applied
    , , ,
    self.data_reordering_options["validation_split"]

```

```

self.hyperparameter["num_epochs"]
self.callbacks_applied, which contains the keys
    PrintEpochInfo
    TensorBoard
    EarlyStopping
    ProgbarLogger
    ReduceLROnPlateau
usually only the first two are used.
'''
if print_dict_before_training:
    self._print_params_as_dictionary()

print("using {0} training samples, which consist of a validation split
      = {1}, begin training for #
      epochs = {2}...".format(
        len(self.data_input["train_feature"].index), self.
        data_reordering_options["
        validation_split"], self.
        hyperparameter["num_epochs"]
      ) )

history = self.model.fit(

    self.data_input["train_feature"],
    self.data_input["train_label"] ,

    epochs = self.hyperparameter["num_epochs"],
    validation_split = self.data_reordering_options["validation_split"]
    ,
    verbose = verbose,
    callbacks = [ self.callback_objects_available[k] for k in self.
                  callbacks_applied ],

)
print("\ntraining complete!\n") # skip a line to avoid overwriting the
                               previous lines.

hist_df = pd.DataFrame(history.history)
epoch_of_interest = -1
if 'EarlyStopping' in self.callbacks_applied:
    epoch_of_interest = hist_df["val_loss"].idxmin()
    self.hyperparameter["num_epochs"] = epoch_of_interest
self.losses.update(dict(hist_df.iloc[epoch_of_interest]))

hist_df['epoch'] = history.epoch # a column handle for plotting
print(hist_df.tail())
self.evaluation_output["hist_df"] = hist_df
self.timing["run_time_seconds"] = time.time() - self.timing["
start_time_raw"]

def auto_generate_session_name(self): # add stuff in front of self.
    session_name
    all_non_dropout_layers = [l for l in self.hyperparameter["hidden_layer"
    ] if type(l) == int]

    num_layer_str = str(len(all_non_dropout_layers)) + "_layer" #
    characterise the session by the
    number of layers used.

    datetime_str = time.strftime("%m%d_%H%M") + "_" # add the date and time

```

```

to prevent name conflict

#Sort these into folders according to their loss values.
'''
loss_value = list(self.evaluation_output["hist_df"]["val_loss"])[-1] #
                                                    get the validation loss from the
                                                    hist_df, which is guaranteed to
                                                    have been generated and
                                                    recorded at the training stage.
if self.losses["loss"]!=0: #if the test loss has been recorded:
    loss_value = self.losses["loss"]
'''
self._evaluate_against_test_set() #force _evaluate_against_test_set to
                                be run so that the self.losses['
                                test_loss'] takes a non-zero (
                                meaningful) value.
rounddown_loss_magnitude = np.floor(np.log10(self.losses['test_loss']))
                                .astype(int) #sort the .png's
                                into folders according to their
                                numbers.
dir_str = "lossabove1e"+ str(rounddown_loss_magnitude) + "/"
if not os.path.exists(dir_str): os.makedirs(dir_str)

# sort by 1. loss value, 2. time,      3.hyperparameter,      4. custome
                                name
session_name = dir_str + datetime_str + num_layer_str + self.
                                session_name

self.session_name = session_name
print("this session's details are saved in", session_name)

def save_params_as_dictionary(self): #Overwrite old *_params.txt dictionary
                                if present
'''save all non-numerical parameters and hyperparameter into a .txt
                                file.'''
original_params_txt = self.session_name.split("layer")[-1]+"_params.txt
                                #always save at the CURRENT
                                working directory; by ignoring
                                all that *layer etc. stuff
                                generated.

f = open(original_params_txt, "w")
f.write("{\n")
def _write_datum(datum):
    if type(datum)==str:
        f.write("'")
        f.write(datum)
        f.write("'")
    else:
        f.write(str(datum))
for k_1 in self.settable_property_list:
    entry = getattr(self, k_1)
    if type(entry)==dict:
        for k_2 in entry:
            f.write(k_2)
            f.write(" : ")
            _write_datum(entry[k_2])
            f.write(" ,\n")
        else:
            f.write(k_1)
            f.write(" : ")

```



```

        _write_datum(entry)
        f.write(" ,\n")
    f.write("}")
    f.close()

def save_NN_weights(self):
    if not os.path.exists(".checkpoints/"): os.makedirs(".checkpoints/") #
                                     make sure .checkpoint/ exist
    self.model.save_weights(".checkpoints/" + self.session_name.split("/")[-1] + ".h5") # save the NN in
                                     the .checkpoints directory,
                                     ignoring the lines before it.

def plot_history(self, show_plot_instead_of_saving = False): # self.
                                     session_name+"_loss_value.png" will
                                     become the name of the saved plot
    num_metrics = len(self.hyperparameter["metrics"])+1 # loss + metrics =
                                     total number of metrics that
                                     will get outputted
    df = self.evaluation_output["hist_df"] #get the hist_df in form of a
                                     shorter variable name.
    columns = df.columns[:-1] #ignoring the last column, which is the epoch
                                     number.
    optimal_epoch = self.hyperparameter["num_epochs"]

    fig, axes = plt.subplots(num_metrics, 1, sharex=True) # Vertically
                                     stack the graphs

    if num_metrics==1:
        axes = [axes,] #wrap the single element into a list so that it can
                                     also be iterated through as
                                     well.

    axes[0].set_title("Performance of the neural network wrt. training
                                     progress")

    for i in range(num_metrics):
        train = columns[i]
        valid = columns[num_metrics+i]
        axes[i].set_ylabel( " ".join(columns[i].replace("squared","sq.").
                                     replace("absolute","abs.").
                                     replace("error","err.").
                                     split("_")) ) #replace the _
                                     with space. and abbreviate.

        axes[i].semilogy(df["epoch"], df[ train ], label="train. error" )
        axes[i].semilogy(df["epoch"], df[ valid ], label="val. error")
        axes[i].legend()
        y_scatt = (df[train][optimal_epoch], df[valid][optimal_epoch])
        axes[i].scatter( np.ones(2)*optimal_epoch, y_scatt, color="r",
                                     marker="x")

    axes[-1].set_xlabel("# epochs")

    if show_plot_instead_of_saving:
        plt.show()
    else:
        plt.savefig(self.session_name + "_error_variation.png")
        plt.clf()
        plt.close()

def _evaluate_against_test_set(self):
    # Print loss values when evaluated against test set
    losses_output = self.model.evaluate(self.data_input["test_feature"],

```

```

self.data_input["test_label"]) #
use tf.model.evaluate to get
the loss values of the
predictions.

if type(losses_output) == list:
    for i in range(len(losses_output)):
        key = list(self.losses.keys())[i]
        self.losses.update({"test_"+key: losses_output[i]})
else:
    self.losses.update({"test_loss": losses_output})
self.save_params_as_dictionary() #overwrite existing dictionary with a
very
print("The loss values and other metrics when evaluated against the
test set are obtained as {0}".
format(self.losses) )

# find the element-wise error
self.evaluation_output["predicted_labels_array_before_post_processing"]
= self.model.predict(self.
data_input["test_feature"]) #use
tf.model.predict to get the
actual prediction themselves.

self._postprocess_output() # populate using the program self.
postprocess_numerical...

self.evaluation_output["error"] = self.evaluation_output["
predicted_labels_array_before_post_processing
"].flatten() - self.data_input["
test_label"].values.flatten()

# use the difference between prediction and true values BEFORE
postprocessing as the deviation/
error list.

# = self.evaluation_output["predicted_labels_array_after_post_processing
"].flatten() - self.data_input["
true_spec"].values.flatten()

# instead of using the difference of their respective values AFTER
postprocessing.

# compute how far off each label is, element-wise
def plot_test_results_histogram(self, show_plot_instead_of_saving=False):
    prepend_in_bracket = ""
    if self.data_preparation_options["log_label"]:
        prepend_in_bracket += "log of "
    if self.data_preparation_options["ft_label"]:
        prepend_in_bracket += "fourier coefficients of "
    if len(self.evaluation_output["error"]) == 0:
        self._evaluate_against_test_set()#ensure that the error list isn't
empty before continuing with
the rest of the current
method"
    plt.hist(self.evaluation_output["error"], bins=25)
    plt.suptitle("Prediction error on each element of the label, (i.e. " +
prepend_in_bracket + "flux PUL"+
")")
    plt.title("loss function(prediction, test_label)={0}".format(self.
losses["test_loss"]))
    plt.xlabel("Error")
    plt.ylabel("Count")
    if show_plot_instead_of_saving:
        plt.show()
    else:
        plt.savefig(self.session_name + "_error_distribution.png")

```

```

plt.clf()
plt.close()

def _postprocess_numerical_values(self, df_or_array, datatype): #datatype
    states whether it's 'label' or 'feature' that's being processed.
    assert (datatype=="feature") or (datatype=="label"), "The datatype must
        be specified either as 'label' or 'feature'."

    if datatype=="feature":
        if self.data_preparation_options["log_feature"]:
            df_or_array = e**df_or_array
    if datatype=="label":
        if self.data_preparation_options["ft_label"]:
            df_or_array = ifft(df_or_array)
        if self.data_preparation_options["log_label"]:
            df_or_array = e**df_or_array
        if not self.data_preparation_options["label_already_in_PUL"]:
            gs = self.data_input["group_structure"].values.flatten()#
            shorten the group structure list into 'gs'

            lethargy_span = np.diff(np.log(gs)) #
            calculate the lethargy span of each bin

            df_or_array = df_or_array*lethargy_span #
            multiply the label (representing flux PUL) by lethargy span to get total flux instead.

    return df_or_array

def _postprocess_output(self):
    self.evaluation_output["predicted_labels_array_after_post_processing"]
        = self.
        _postprocess_numerical_values(
            self.evaluation_output["
                predicted_labels_array_before_post_processing"], "label")

    self.data_input["true_spec"] = self._postprocess_numerical_values(self.
        data_input["test_label"], "label") #un-log and un-fourier
        transform the data to get it back into the correct form.

def _convert_to_PUL(self, flux):
    gs = self.data_input["group_structure"].values.flatten() #shorten the
        variable name into 'gs'
    lethargy_span = np.diff(np.log(gs)) #calculate the lethargy span of
        each bin

    flux = flux/lethargy_span
    return flux

def _split_line_at_threshold(self, flux, upper_or_lower = "lower",
    threshold = 2):
    convert flux to flux PUL, and chop it, leaving only the half that's above/below the threshold energy value.
    gs = self.data_input["group_structure"].values.flatten()

```

```

# flux = self._convert_to_PUL(flux) #the flux has already been
                                    converted to PUL when inputting
                                    it.
thres_ind = abs(gs - threshold).argmin() #find the index of the closest
                                          to the threshold

if upper_or_lower == "lower":
    gs_cut = gs[:thres_ind+1]
    flux_cut = np.hstack([flux[0], flux[:thres_ind]])
elif upper_or_lower=="upper":
    gs_cut = gs[thres_ind:]
    flux_cut = np.hstack([flux[thres_ind], flux[thres_ind:]])
return gs_cut, flux_cut

def _side_by_side_plot(self, press, ind, true_line, predicted_line ,
                        ref_spec_line=None, ref_info_line=
                        None):
    '''
    make two plots,
        ax1 compares total flux in each bin according to bin number, by
        plotting predicted flux and
        true_flux side-by-side

        ax2 plots the flux in each bin.
    '''
    fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
    # label the axes.
    ax1.set_xlabel("bin number"); ax1.set_ylabel("flux per unit lethargy
        per unit fluence (1/s)")
    ax2.set_xlabel("energy (MeV)"); ax2.set_ylabel("flux (per unit
        lethargy)")
    # make the plot on the right a log-log plot,
    # ax1.set_yscale("log")
    ax2.set_yscale("log"); ax2.set_xscale("log")

    #add titles
    ax1.set_title("smooth plot of spectrum for comparison purpose."); ax2.
        set_title("log-log plot of
        spectrum")
    plt.suptitle("test spectrum " + str(ind))
    if type(ref_info_line)!=type(None):
        plt.suptitle(ref_info_line["title"]) #overwrite the supitle
    # link up to the press() function (defined locally within the scope of
        self.compare_individual_spectra
        ())
    fig.canvas.mpl_connect('key_press_event', press)
    #actual plotting
    ax2.step(*self._split_line_at_threshold(true_line, "upper", threshold=0
        ), label="true fluence", alpha=0
        .8)
    ax2.step(*self._split_line_at_threshold(predicted_line, "upper",
        threshold=0), label="fluence
        predicted by NN", alpha=0.8)
    ax1.semilogy(true_line, label="true fluence", alpha=0.8)
    ax1.semilogy(predicted_line, label="fluence predicted by NN", alpha=0.8
        )
    #plotting the original spectrum if it exist.
    if type(ref_spec_line)!=type(None):
        ax2.step(*self._split_line_at_threshold(ref_spec_line), label="
        original flux before
        perturbation", alpha=0.8)
        ax1.semilogy(ref_spec_line, label="original flux before

```

```

                                                                    perturbation", alpha=0.8)

#apply legends
ax1.legend()
ax2.legend()
#maximize window
mng = plt.get_current_fig_manager()
if hasattr(mng, 'frame'): # works with ubuntu
    mng.frame.Maximize(True) # try to maximize the window
else:
    try:
        mng.window.showMaximized()
        # mng.resize(*mng.window.maxsize())
    except:
        pass # ignore this if python cannot maximize window; it has to
              # be maximized manually.

plt.show()
plt.clf(); plt.close() #show an then close.

def _C_E_plot(self, press, ind, true_line, predicted_line, ref_spec_line=
               None, ref_info_line=None, threshold=
               2):
    '''
    Plot the original spectrum and the NN's prediction on the same graph,
    and show the C/E value of each
    point below it.
    '''
    # naming axes according to the scale at x and y axes.
    fig, ([log_data, lin_data],
          [log_ce,      lin_ce]) = plt.subplots( 2, 2, sharex='col', sharey=
          'row',
          figsize=(12, 8),
          gridspec_kw={

```

```

plt.suptitle("test spectrum " + str(ind))
if type(ref_info_line)!=type(None):
    plt.suptitle(ref_info_line["title"])
log_data.set_xscale("log")
lin_data.set_xscale("linear")
log_data.set_yscale("log")
log_data.set_ylabel("flux per unit lethargy per unit fluence (1/s)")
log_ce.set_ylabel("calculated/expected (C/E)")
unit="eV"
if threshold<100: unit="MeV"
log_ce.set_xlabel("E ({0}) on log scale".format(unit) )
lin_ce.set_xlabel("E ({0})".format(unit) )
log_ce.axhline(1,color="gray")
lin_ce.axhline(1,color="gray")

def plot_data(flux, label):
    log_data.step(*self._split_line_at_threshold(flux, "lower",
                                                threshold), label=label,
                                                alpha=0.8)
    lin_data.step(*self._split_line_at_threshold(flux, "upper",
                                                threshold), label=label,
                                                alpha=0.8)

def plot_ce(ce):
    log_ce.scatter(*self._split_line_at_threshold(ce, "lower",
                                                threshold), marker="x",
                                                alpha=0.6) #fmt="C0x"
    lin_ce.scatter(*self._split_line_at_threshold(ce, "upper",
                                                threshold), marker="x",
                                                alpha=0.6) #fmt="C0x"

plot_data(predicted_line, label="fluence predicted by NN")
plot_data(true_line, label="true fluence")
if type(ref_spec_line)!=type(None):
    plot_data(ref_spec_line, label="ref_spec_line_before_perturbation")
    #overwrite the suptitle

plot_ce(predicted_line/true_line)

# add legend to the graph
log_data.legend()
fig.tight_layout(rect=[0, 0, 1, 0.95]) # top right hand corner of 'rect
# has the coordinate (1,0.95) to prevent the suptitle clipping into the
graph
plt.savefig(self.session_name + "_test_" + str(ind).zfill(3) + "
            _fluence.png", dpi=180)

plt.clf()
plt.close()

def _reaction_rate_compare(self, press, ind, true_line, predicted_line ,
                        ref_spec_line=None, ref_info_line=
                        None, save_or_not=True):
    if type(ref_spec_line)!=type(None):
        ref_spec_line = np.array(ref_spec_line)

    response_matrix = np.array(self.data_input["response_matrix"])
    assert np.ndim(response_matrix)==2, "Please load the response matrix
        before doing self.
        _reaction_rate_compare()!"
    true_activities = response_matrix.dot(true_line)

```

```

predicted_activities = response_matrix.dot(predicted_line)
num_activites = np.arange( len(response_matrix) )

dist_in_log_space = np.log(predicted_activities/true_activities)
mu = 0
sigma = sum(np.sqrt( (dist_in_log_space-mu)**2 /len(dist_in_log_space)
                    ))
# chi2_dof = sum( (dist_in_log_space-0)**2 )/len(dist_in_log_space)
# chi2txt = r"total $\frac{\chi^2}{DoF}$="+ str(chi2_dof) +"\n"+"
#                                     assuming C/E is lognormally
#                                     distributed around 1."

if save_or_not:
    fig, (bar, ce) = plt.subplots(2,1, sharex=True,
                                gridspec_kw={'height_ratios': [6, 1]})

    reaction_names = [ i.replace("_","") for i in self.data_input["
                        response_matrix"].index ]

    ce.set_xticks(num_activites)
    ce.set_xticklabels(reaction_names, rotation=30, fontdict={"fontsize
                                                                ":8})

    numBars = 2
    if type(ref_spec_line)!=type(None):
        ref_activites = response_matrix.dot(ref_spec_line)
        numBars = 3
    width = 0.8/numBars

    bar.set_ylabel("activity per unit fluence(1/s)")
    bar.bar(num_activites + width , predicted_activities, label="
                                                    activities predicted by NN",
            width=-width, align="edge")
    bar.bar(num_activites , true_activities, label="true activities",
            width=-width, align="edge")
    if type(ref_spec_line)!=type(None):
        bar.bar(num_activites + 2*width, ref_activites, label="
                                                    original activities",
            width=-width, align="
            edge")

    bar.legend()
    bar.set_yscale("log")

    ce.axhline(1,color="gray")
    ce.scatter(num_activites , predicted_activities/true_activities,
               marker="x")
    ce.set_ylabel("C/E")

    sigmatxt = r"$\sigma$="+ str(sigma) +"\n"+"assuming C/E is normally
                                                    distributed in log space,
                                                    with a mean of 0."

    bar.set_title(sigmatxt)

    plt.suptitle( "test spectrum " + str(ind) )
    if type(ref_info_line)!=type(None):
        plt.suptitle(ref_info_line["title"]) #overwrite the supitle

# fig.text( 0.5, 0.0 , chi2txt, va="bottom", ha="center")
# link up to the press() function (defined locally within the scope
# of self.
# compare_individual_spectra()

```

```

        )
        fig.tight_layout(rect=[0, 0.03, 1, 0.95])
        plt.savefig(self.session_name + "_test_" + str(ind).zfill(3) + "
                    _activities.png", dpi=100)

        plt.clf()
        plt.close()
    return sigma

def _renormalize_prediction(self, fluxPUL):
    if self.hyperparameter["loss_func"]=="mean_pairwise_squared_error":
        n = np.ndim(fluxPUL)
        fluxPUL = ((fluxPUL).T/np.sum(fluxPUL, axis=n-1)).T
    return fluxPUL

def compare_individual_spectra(self, using_simple_data=False, threshold = 2
                               , save_C_E_plots = True,
                               save_reaction_rate_comparisons=True,
                               silent_mode=False):

    def press(event): #for stopping the plot comparison program when the
                      key 'q' is pressed

        if event.key == 'q':
            self.keep_showing_figure = not self.keep_showing_figure
            print("Pressed 'q' to toggle self.keep_showing_figure to {0}".
                  format(self.
                          keep_showing_figure))

    does_ref_spec_exist = not (type(self.data_input["ref_spec"]) == type(
                                None))

    # Need to compare the self.data_input["true_spec"] against the
    evaluation_output["
    predicted_labels_array_after_post_process
    "].

    # Therefore the next part gets the evaluation_output["
    predicted_labels_array_after_post_process
    "]

    if type(self.evaluation_output["
    predicted_labels_array_after_post_process
    "])==type(None): #in case the
    _evaluate_against_test_set hasn'
    t been ran

        #(such that _postprocess_output hasn't been called to populate
        evaluation_output properly)
        print("postprocessing test_label and predicted_labels to get
              true_spec and predicted
              spectrum respectively.")

        self._evaluate_against_test_set()

    #shorten the names
    true_spec = self.data_input["true_spec"].values
    predicted_labels = self.evaluation_output["
    predicted_labels_array_after_post_process
    "]

    if does_ref_spec_exist:
        ref_spec = self.data_input["ref_spec"].values
        ref_info = self.data_input["ref_info"]
        does_ref_info_exist = not type(ref_info)==type(None)

    #covnert to PUL if not already in PUL.
    if (not using_simple_data) and (not self.data_preparation_options["

```



```

                                label_already_in_PUL"]):
predicted_labels = self._renormalize_prediction(self.
                                                _convert_to_PUL(
                                                predicted_labels))
true_spec = self._renormalize_prediction(self._convert_to_PUL(
                                                true_spec))#The true
                                                spectrum is left in the raw,
                                                non-PUL state until now.

if does_ref_spec_exist:
    ref_spec = self._renormalize_prediction(self._convert_to_PUL(
                                                ref_spec))

sigma_list = []
for ind in range(len(predicted_labels)):
    if using_simple_data:
        fig, ax1 = plt.subplots()
        ax1.bar(np.arange(5), true_spec[ind], label="true fluence",
                width=0.4)
        ax1.bar(np.arange(5) + .4, predicted_labels[ind], label="
                fluence predicted by NN"
                , width=0.4)

        ax1.legend()
        plt.suptitle("test spectrum " + str(ind))
        fig.canvas.mpl_connect('key_press_event', press)
        # link up to the press() function (defined locally within the
        scope of self.
        compare_individual_spectra
        ())

        plt.show()
        plt.clf(); plt.close()
    else:
        ref_spec_line = None
        if does_ref_spec_exist:
            ref_spec_line = pd.DataFrame(ref_spec).iloc[ind]

        ref_info_line = None
        if does_ref_info_exist:
            ref_info_line = pd.DataFrame(ref_info).iloc[ind]
        if not silent_mode:
            self._side_by_side_plot(press, ind, true_spec[ind],
                                    predicted_labels[ind], ref_spec_line=
                                    ref_spec_line,
                                    ref_info_line=
                                    ref_info_line)

        if save_C_E_plots:
            self._C_E_plot(press, ind, true_spec[ind], predicted_labels
                            [ind], ref_spec_line
                            =ref_spec_line,
                            ref_info_line=
                            ref_info_line,
                            threshold=threshold)

        sigma = self._reaction_rate_compare(press, ind, true_spec[ind],
                                                predicted_labels[ind],
                                                ref_spec_line=
                                                ref_spec_line,
                                                ref_info_line=
                                                ref_info_line,
                                                save_or_not =
                                                save_reaction_rate_comparisons
                                                )

```

```

        #will not save if save_reaction_rate_comparisons is False; in
        #which case it will
        #simply return the sigma
        #to be appended to the
        #sigma_list below:

        sigma_list.append(sigma)
    if not self.keep_showing_figure:
        break #condition to stop showing more figures if 'q' is pressed
        (self.
         keep_showing_figure is
         set by the locally
         defined function 'press
         ')

mean_sigma = np.mean(sigma)
self.losses.update({'std_of_log_of_C_over_E_reaction_rates':mean_sigma})
'''
#THIS IS A BODGE to insert a line into the _params.txt.
if not save_C_E_plots:
    original_params_txt = self.session_name.split("layer")[-1]+"_params
    .txt"

    with open(original_params_txt,"r") as f:
        lines = f.readlines()
    for i in range(len(lines)):
        if "}" in lines[i]:
            brace_line_num = i
    with open(original_params_txt,"w") as f:
        [ f.write(l) for l in lines[:brace_line_num] ]
        f.write('std_of_log_of_C_over_E_reaction_rates : '+str(
            mean_sigma)+' ,\n')
        [ f.write(l) for l in lines[brace_line_num:] ]
'''
self.save_params_as_dictionary()

def predict_from_additional_file(self, prediction_file_name):
    raw_unlabelled_features = pd.read_csv(prediction_file_name, header=None
        , comment="#")
    processed_unlabelled_features = self._preprocess_numerical_values(
        raw_unlabelled_features, "
        features")
    prediction_label_array_before_post_processing = self.model.predict(
        processed_unlabelled_features)
    return self._postprocess_numerical_values(
        prediction_label_array_before_post_processing, "feature")

def plot_training_spectra(self, threshold):
    processed_train_label_df = pd.DataFrame(self.data_input["train_label"])
    max_num_plots=None
    if len(processed_train_label_df)>200: max_num_plots=50

    fig, (log_data, lin_data) = plt.subplots( 1, 2, sharey=True,
        figsize=(12, 7),
        gridspec_kw={'width_ratios'

```

```

log_data.set_xscale("log")
lin_data.set_xscale("linear")
log_data.set_yscale("log")
log_data.set_ylabel("flux per unit lethargy per unit fluence (1/s)")
unit="eV"
if threshold<100: unit="MeV"
log_data.set_xlabel("E ({0}) on log scale".format(unit))
lin_data.set_xlabel("E ({0})".format(unit))

for flux in processed_train_label_df.iloc[:max_num_plots].iterrows():
    log_data.step(*self._split_line_at_threshold(flux[1], "lower",
                                                threshold=threshold), alpha=
                                                0.4)
    lin_data.step(*self._split_line_at_threshold(flux[1], "upper",
                                                threshold=threshold), alpha=
                                                0.4)

plt.suptitle("Some of the spectra used to train the neural network with
            ")

plot_name = self.session_name+"_training_spectra"
if hasattr(self, "train_label_file"): plot_name = ".".join(getattr(self
, "train_label_file").split(".")
[:-1])

plt.savefig(plot_name+".png")

```

## B Neural network abstractions and controller

The following contains the higher level abstractions, as well as functions which walks the user through the process of creating a neural network interactively.

neuralnetworktrainer.py

```

from neuralnetworklibrary import *
from matplotlib import pyplot as plt
#This files contains the toolsets for doing the following three things:
#1. To demonstrate that neural network works when using_simple_data
# (i.e. using the 5 reaction_rates obtained by folding the 5 randomly
generated flux values through a 5x5 non-
singular matrix, therefore giving a
fully determined problem.)
#2. To demonstrate the neural network works when trying to unfold the simulated
data.
#3. To investigate what hyperparameters is required if we were to invert the
real data.
'''
#####Higher level automations#####
This program offers two warpper method, which does all of the above methods all
at once:

run_real_spectra / run_demo
'''
class NeuralNetworkHandler(NeuralNetwork):
    def __init__(self):
        super().__init__()
        self.using_simple_data=False # assume, by default, that we're not
reading the simple, 5x5 case

```

```

data.

self.reactor_prefix=""
self.activation_system=""

def read_demo_data(self, using_simple_data=False):
    self.using_simple_data = using_simple_data
    if self.using_simple_data:
        print("using simple, 5x5, non-singular (fully determined) data ...")
        self.reactor_prefix="simple_"
        self.activation_system=""
        self.data_preparation_options["label_already_in_PUL"] = True
    else:
        self.reactor_prefix="GS_eq_1_JAEA_FNS_"
        self.activation_system="ACT_"
    feature_file= self.reactor_prefix + self.activation_system + "RR.csv"
    label_file = self.reactor_prefix + "spectra.csv"
    ref_spec_file=self.reactor_prefix + "reference_spectra.csv"
    response_matrix_file=self.reactor_prefix+self.activation_system+"
        Response_Matrix.csv"

    gs_file = "demo_gs.csv"
    self.load_data(feature_file, "feature_before_preprocessing")
    self.load_data(label_file, "label_before_preprocessing")
    if not self.using_simple_data:
        self.load_data(ref_spec_file, "ref_spec")
        self.load_data(gs_file, "group_structure")
        self.load_data(response_matrix_file, "response_matrix")

# higher level methods: methods that uses other lower level methods; read
# these for a summary of the program
def cast_and_preprocess_data(self):
    """
    Condense the whole data preparation stage into a single, more compact
    method.

    First trim the data (according to the self.data_re_ordering_options)
    """
    # read the raw data
    if type(self.data_input["train_feature"])==type(None): #only do the
        # splitting and shuffling if the
        # train/test sets haven't been
        # populated yet.

        self.trim_data() # trim the data
        self.shuffle() # shuffle the DataFrames
        self.split_into_sets() # split the DataFrames into training sets
        # and testing sets.
    self.preprocess_input() # Take log and fourier analyse

def build_and_train_model(self, quietly=False):
    self.build_model(print_pretty_logo = not quietly)
    self.train_model(print_dict_before_training = not quietly)

def plot_performance(self, show_plot_instead_of_saving=False):
    self.plot_history(show_plot_instead_of_saving=
        show_plot_instead_of_saving)
    self.plot_test_results_histogram(show_plot_instead_of_saving=
        show_plot_instead_of_saving)

def show_results_of_training(self): #without saving
    # plot and save its performance
    self.plot_performance(show_plot_instead_of_saving = True)

```

```

#Examine the weight matrix (as compared to the weights matrix)
self.compare_individual_spectra(using_simple_data=self.
                                using_simple_data,
                                save_C_E_plots= False)

def compare_with_known_inverse(self, response_matrix_file_name="
                                demogenerator/simple_response_matrix
                                .csv"):
    '''
    Compare the weights obtained for the linear regresser
    against the inverse of the non-singular matrix for the simplecase.
    '''
    assert self.using_simple_data, "This method is only used for the 5x5
                                    fully-determined case!"

    weights, biases = read_NN_weights(self.session_name)
    response_matrix = np.matrix(pd.read_csv(response_matrix_file_name,
                                              header=None))

    # Compare the analytically obtained inverse with the weights matrix
    import seaborn as sns
    sns.heatmap(response_matrix.I.T, annot=True)
    plt.savefig("true_inverse.png")

    plt.cla(); plt.clf(); plt.close() #clear everything

    sns.heatmap(weights["layer_1"], annot=True)
    plt.savefig("NN_weights_emulating_inverse_matrix.png")
    print("See the newly saved *.png ('true_inverse' and '
                                NN_weights_emulating_inverse_matrix
                                ') to compare how well the NN
                                emulated the weights matrix of
                                the 1st layer")

    print("Additionally, the biases in the 1st layer are \n", biases["
                                layer_1"])
    return response_matrix, weights["layer_1"], biases["layer_1"]

def save_metrics_and_compare_reproducibly(self, threshold, save_plots=True,
                                           silent_mode=False):
    self.auto_generate_session_name()
    self.save_NN_weights()
    self.save_params_as_dictionary()
    self.plot_performance()
    if self.using_simple_data:
        #Examine the weight matrix (as compared to the response matrix's
                                inverse)
        self.compare_with_known_inverse()
        self.compare_individual_spectra(using_simple_data=True)
    else:
        # opening up each predicted spectrum and plotting it side-by-side
                                with the true spectrum and
                                original spectrum
        self.compare_individual_spectra(threshold=threshold, save_C_E_plots
                                        =save_plots,
                                        save_reaction_rate_comparisons
                                        =save_plots, silent_mode=
                                        silent_mode)

def run_demo(self, using_simple_data=False):
    self.using_simple_data = using_simple_data
    self.read_demo_data()
    self.cast_and_preprocess_data()

```

```

self.build_and_train_model()
self.save_metrics_and_compare_reproducibly(threshold=2)

def run_real_spectra(self, save_plots=True, silent_mode=False):
    #self.condense_into_one_csv(directory)
    self.cast_and_preprocess_data()
    self.build_and_train_model()
    self.save_metrics_and_compare_reproducibly(threshold=2E6, save_plots=
                                                save_plots, silent_mode =
                                                silent_mode)

##Tutorials for new users

def program_structure(self):
    print("# To run a process successfully, the following methods have to
        be run:")

    print("# 0. Setting hyperparameters")
    print("interactive_neural_network_maker #alternatively, these can be
        changed manually by using
        setattr().")

    print("# 1. load and pre-processing")
    print("load_data('feature_before_preprocessing', '
        label_before_preprocessing', '
        group_structure')")

    print("# or in case of using demo data:")
    print("read_demo_data")
    print("trim_data (optional)")
    print("shuffle (optional)")
    print("split_into_sets (can skip if data is directly loaded into '
        train_*' and 'test_*' instead of
        splitting from '*'
        _before_preprocessing' in the
        load_data() step)")

    print("preprocess_input")
    print("    # All of section 1 above, except load_data, is summarized by
        the method of
        cast_and_preprocess_data.")

    print("# 2. build and train model")
    print("build_model")
    print("train_model")
    print("    # These are summarized by build_and_train_model in
        NeuralNetworkHandler")

    print("# 3. for saving data (optional)")
    print("auto_generate_session_name")
    print("save_params_as_dictionary")
    print("save_NN_weights")
    print("# 4. for plotting (optional)")
    print("plot_history")
    print("plot_test_results_histogram")
    print("compare_individual_spectra")
    print("    # 3 and 4 are summarized by show_results_of_training and
        save_metrics_and_compare_reproducibly
        in NeuralNetworkHandler")

    print("")
    print("##### Alternatively section 1-4 above can be replaced by the
        single method")
    print("run_demo # for running the demonstrative data")
    print("# or in case of running a real data:")
    print("run_real_spectra")

```

```

def input_instructions(self):
    print("The data are inputted in the form of csv's,")
    print("each row representing one spectrum or its reaction rate.")
    print("The file containing the spectra should be loaded as the
            feature_file;")
    print("while the file containing the corresponding reaction rates
            should be loaded as the
            label_file.")
    print("A single line csv (horizontal or vertical) containing all the
            boundaries of the bins should be
            loaded as the gs_file")

def tutorial_demo(self):
    # print("This interactive tutorial is designed to be used in an
            interactive python environment (
            e.g. ipython).")
    print("This method walks the user through the process of creating a
            neural network, and then trains
            and runs this neural network on
            the demo data.")

    print("\n")
    print("The following is a list of options and hyperparameters to be
            inputted into the neural network
            .")

    self.interactive_neural_network_maker()
    print("Please state whether you would like to use the 5x5 fully-
            determined case, or the
            simulated 11x171 response matrix
            case.")

    while True:
        using_simple_data_y_n = input("type 'y' for fully-determined case,
            'n' for 11x171")

        if using_simple_data_y_n=="y":
            using_simple_data=True
            break
        elif using_simple_data_y_n=="n":
            using_simple_data=False
            break
    print("running the demo...")
    self.run_demo(using_simple_data=using_simple_data)

def interactive_menu(self):
    '''start here'''
    print("0. program_structure")
    print("1. input_instructions")
    print("2. tutorial_demo")
    print("3. interactive_neural_network_maker (to set the options and
            hyperparameters for this neural
            network)")

    while True:
        x = input("choose a number from the menu above:")
        if x in [str(i) for i in range(4)]:
            break
        print("input not accepted.")
    print("

            -----
            ")

    if x=="0":
        self.program_structure()
    elif x=="1":

```

```

        self.input_instructions()
    elif x=="2":
        self.tutorial_demo()
    elif x=="3":
        self.interactive_neural_network_maker()

def continuous_neural_network_runner(filename, demo=False, using_simple_data=False):

    import shutil as shu
    print("_"*shu.get_terminal_size().columns) # print a separation line
                                                # between each run.

    first_dict_lines, rest_of_the_lines = cut_file_in_halves(filename)
    dictionary_read = convert_lines_to_dict(first_dict_lines)
    #instantiate a NeuralNetworkHandler()
    nn = NeuralNetworkHandler()
    for k, v in dictionary_read.items():
        print(k, ":", v)
        if k.endswith("_file"):
            assert k[:-5] in nn.data_input.keys(), "File type not found"
            setattr(nn, k, v)
            nn.settable_property_list.append(k)
            nn.load_data(v, k[:-5])
        else:
            nn.try_to_update_attribute(k,v)
    #overwrite only if the attributes are set without raising any errors.
    overwrite_file_by_removing_first_dict(filename, rest_of_the_lines)
    if demo:
        nn.run_demo(using_simple_data=using_simple_data)
    else:
        nn.run_real_spectra(save_plots=False, silent_mode=True)

if __name__=="__main__":
    if len(sys.argv)==1:
        while True:
            continuous_neural_network_runner("real_hyperparameter_tweaking.txt",
                                                demo=False)

        while False:
            continuous_neural_network_runner("pre-presentation-demos.txt", demo
                                                =True)

    elif len(sys.argv)>1:
        try:
            int(sys.argv[1])
            while True:
                continuous_neural_network_runner("job_number_"+sys.argv[1]+".
                                                txt", demo=False)

        except ValueError:
            filename = "real_hyperparameter_tweaking.txt"
            if sys.argv[1]=="debug":
                first_dict_lines, rest_of_the_lines = cut_file_in_halves(
                    filename)

                dictionary_read = convert_lines_to_dict(first_dict_lines)
                nn = NeuralNetworkHandler()
                for k,v in dictionary_read.items():
                    print(k,":",v)
                    if k.endswith("_file"):
                        assert k[:-5] in nn.data_input.keys(), "File type not
                                                                    found"

                        setattr(nn, k, v)
                        nn.settable_property_list.append(k)
                        nn.load_data(v, k[:-5])

```



```

        else:
            nn.try_to_update_attribute(k,v)
            overwrite_file_by_removing_first_dict(filename,
                                                    rest_of_the_lines)
            nn.run_real_spectra(save_plots=True, silent_mode=False)
            nn.plot_training_spectra(threshold=2e6)

```

## C Code for benchmarking

This code uses 'unfoldingsuite', which contains implementations of MAXED and GRAVEL in python, developed locally at CCFE, to unfold spectra from various a priori. Their performance can then be used as benchmarks for the neural network unfolding results to be compared against.

comparison\_with\_existing.py

```

import numpy as np
import pandas as pd
import shutil
from unfoldingsuite.datahandler import UnfoldingDataHandler_2
from unfoldingsuite.nonlinearleastsquare_2 import SAND_II_2, GRAVEL_2
from unfoldingsuite.maximumentropy_2 import MAXED_2
from unfoldingsuite.parameterised_2 import Parameterised_2
'''
from unfoldingsuite.tools.unfolding_data_handler import UnfoldingDataHandler
from unfoldingsuite.nonlinearleastsquares.sand2 import SAND_II
from unfoldingsuite.nonlinearleastsquares.gravel import GRAVEL
from unfoldingsuite.maximumentropy.maxed import MAXED
'''

def conver_to_PUL(vector, group_structure):
    assert len(group_structure)-1==len(vector), "must have N+1 boundaries for
                                                    vector length N={0}, but instead {1}
                                                    boundary values are found".format(
                                                    len(vector), len(group_structure))
    leth_span= np.diff(np.log(group_structure))
    return vector/leth_span

DATASET="fusion_test"
A_PRIORI_IS_FLAT=True
# true_spec_list = pd.read_csv("../real_"+DATASET+"_normed.csv",header=None)
reaction_rates_list = pd.read_csv("../real_"+DATASET+"_normed_ACT.csv",header=
None)
response_matrix = pd.read_csv("../response_matrix_ACT_175_gs.csv", header=None,
index_col=[0])
group_structure = pd.read_csv("../175_gs.csv",header=None).values.flatten()

maxed_solution=[]
gravel_solution=[]

for i in range(len(reaction_rates_list.index)):
    rr_line = reaction_rates_list.iloc[i]
    unfold = UnfoldingDataHandler_2()
    unfold.set_vector('reaction_rates', list(rr_line) )
    unfold.set_vector_uncertainty('reaction_rates', np.full( len(rr_line),0.
05 ).tolist() )
    unfold.set_matrix('response_matrix', response_matrix.values)
    # unfold.load_vector('a_priori')
    if A_PRIORI_IS_FLAT:
        unfold.set_vector('a_priori', np.ones( np.shape(unfold.get_matrix('
response_matrix'))[1] ).tolist()

```

```

        )# set flux PUL a priori to be
        a flat spectrum, dimension =
        number of energy bins.
else:
    unprocessed_a_priori = pd.read_csv("real_"+DATASET+"_a_priori.csv",
                                       header=None).values[i]# find the
                                       i-th line of the a priori in
                                       the a priori file.

    a_priori = conver_to_PUL(unprocessed_a_priori, group_structure)
    unfoldr.set_vector('a_priori',a_priori.tolist())
    gravel=GRAVEL_2(verbosity=0)
    gravel.set_all_parameters(unfoldr)
    try:
        gravel.run('n_trials', [10000]) #run until we reach num_trials = 1000
    except:
        break
    gravel_solution.append(gravel.get_vector('solution'))
    print("finished line", i)

    maxed=MAXED_2()
    maxed.set_all_parameters(unfoldr)
    maxed.run('basin_hopper',[ ]) #empty list to denote use all default
                                parameters of the basin hopper
                                algorithm.

    maxed_solution.append(maxed.get_vector('solution'))
if A_PRIORI_IS_FLAT:
    np.savetxt("real_"+DATASET+"_gravel_"+ "flat"+"_a_priori_solution.csv",
              gravel_solution, delimiter=",")
    np.savetxt("real_"+DATASET+" "_maxed_"+" "flat"+"_a_priori_solution.csv",
              maxed_solution, delimiter=",")
else:
    np.savetxt("real_"+DATASET+"_gravel_"+"nn"+"_a_priori_solution.csv",
              gravel_solution, delimiter=",")
    np.savetxt("real_"+DATASET+" "_maxed_"+" "nn"+"_a_priori_solution.csv",
              maxed_solution, delimiter=",")

ref_info_file = "real_"+DATASET+"_normed_ref_info.csv"
shutil.copyfile("../"+ref_info_file, ref_info_file)

```

## D Fully determined simulation data generation

Creates a 5 energy-bins fluence vector, which is then folded through a  $5 \times 5$  response matrix; both of which are randomly generated. Each element both were picked from a uniform random distribution larger than 1. The upper bound of the elements in the vector were chosen as 15 and the upper bound of the elements in the response matrix were chosen to be 50.

simple\_non\_singular\_case.py

```

from unfoldingsuite.nonlinearleastquares.gravel import GRAVEL
import numpy as np

SIZE = 5                                # Shape of square response matrix =(SIZE x
                                       SIZE)
RESPONSE_RANGE = (1.0, 50.0)           # Range response matrix values can take
FLUX_RANGE = (1.0, 15.0)                # Range flux values can take
NUMBER_OF_SPECTRA = 100                # For training the neural network

# Generate a random response matrix, checking that it is full rank.

```

```

response = np.matrix([np.zeros(5) for row in range(SIZE)])

np.random.seed(0) #Make sure we get the same response matrix every time.

while np.isinf(np.linalg.cond(response)):#Make sure that the response matrix is
    not singular.
    for row in range(SIZE):
        for col in range(SIZE):
            response[row, col] = RESPONSE_RANGE[0] + (np.random.rand() * (
                RESPONSE_RANGE[1] -
                RESPONSE_RANGE[0]))

# Generate random spectra, and fold into reaction rates

def generate_N_spectra_and_reaction_rates(N):
    spectra, reaction_rates = [], []
    for spectra_index in range(N):
        spectrum = np.matrix([np.zeros(1) for row in range(SIZE)])
        for row in range(SIZE):
            spectrum[row] = FLUX_RANGE[0] + (np.random.rand() * (FLUX_RANGE[1]
                - FLUX_RANGE[0]))

        spectra.append(spectrum)
        reaction_rates.append(response * spectrum)
    print(np.shape(spectra))
    return np.reshape(spectra, [-1,SIZE]), np.reshape(reaction_rates, [-1,SIZE])

# Print out random spectra, their response functions and check that the inverse
    can be found

print("R=", response)
if __name__=="__main__":
    np.savetxt("for_test_spectra.csv", response, delimiter=",") #Saving this
        response matrix just for reference
        purpose.

    spectra, reaction_rates = generate_N_spectra_and_reaction_rates(
        NUMBER_OF_SPECTRA)

    np.savetxt("../simple_spectra.csv", spectra, delimiter=",") #Features
    np.savetxt("../simple_RR.csv", reaction_rates, delimiter=",") #Labels

```

## E Underdetermined simulation data generation

For each of the 14 FISPACT reference spectra, each is parametrised into a list of peaks. The height of these peaks were then perturbed to form a ‘new’ spectrum. This ‘new’ spectrum is then folded through a corresponding response matrix.

spectrumrandomizer.py

```

import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import warnings
from numpy import sqrt, pi, exp, log
import copy
import time
start_time = time.time()

def reshape_data_for_smooth_spectrum_plotting(bin_boundaries, bin_heights,
    log_scale=False): #For SMOOTH plotting,

```

```

                                for easier visualization in linear scale
                                , since histogram-like graphs looks ugly
                                in non-.

from scipy.stats.mstats import gmean
'''
reshape data into a format such that, when plugged into
plt.plot(x,y), gives a smooth plot.
The reshape_data_for_histogrammic_spectrum_plotting function defined above
                                gives very jagged-edges;
in contrast, this reshape_data_for_smooth_spectrum_plotting function is
                                equivalent to applying anti-aliasing
                                technique on the spectrum,
smoothing out the spectrum.
'''

assert len(bin_heights)+1==len(bin_boundaries)
bin_boundaries=np.hstack(np.array(bin_boundaries))
#If plotting on a linear x-axis scale, the arithmetic mean is used as the
                                class mark.
class_marks=bin_boundaries[:-1]+np.diff(bin_boundaries)/2
if log_scale:
    #If plotting on a log-x scale, the geometric mean is used to find the
                                class mark instead.
    class_marks= gmean([bin_boundaries[:-1], bin_boundaries[1:]])
return class_marks, bin_heights #return the x,y values requiried

def reshape_data_for_histogrammic_spectrum_plotting(bin_boundaries, bin_heights)
    :
    '''
    reshape data into a format such that, when plugged into
    plt.plot(x,y), gives a histogram-like, square-edges plot.
    i.e. uniform height within each bin.
    '''
    assert len(bin_heights)+1==len(bin_boundaries)# The bin_boundaries variable
                                includes both upper and lower
                                bounds for each bin
    bin_boundaries,bin_heights=np.array(bin_boundaries),np.array(bin_heights)
    Intercalation = np.repeat(np.arange(len(bin_boundaries)), 2)[:-1] #indices
                                to be used in the next line.
    return bin_boundaries[Intercalation[1:]],bin_heights[Intercalation[:-1]] #
                                return the x, y values requiried

def get_group_structure(reactor):
    '''read a csv of the correct name in the same directory'''
    group_structure_csv_suffix="_Group_Structure.csv"
    return np.genfromtxt(reactor+group_structure_csv_suffix,delimiter=",")

def convert_to_centroid_values(energy_group_structure):
    from scipy.stats.mstats import gmean
    class_marks= gmean([energy_group_structure[:-1], energy_group_structure[1:]
                        ])
    return class_marks

def preprocess_df(df, n_sample=1, keep_fixed_fraction=0.6):
    distilled_df = df[["distribution","a_true","b_true","amplitude_true"]] #
                                only extract the four values that
                                matters

    num_func = len(df.index)
    output_df_list = []
    numTrues = int(np.round(keep_fixed_fraction*num_func))
    numFalses= num_func-numTrues

```

```

#Duplicate it up to n_sample of them
for n in range(n_sample): #The following loop can be sped up by using numpy
                             arrays better and perhaps storing
                             the data as a dataframe instead of a
                             list.

    keep_fixed_bool_vector = np.random.choice( [True,]*numTrues + [False,]*
                                                numFalses , size=num_func,
                                                replace=False)

    new_df = distilled_df.copy()
    new_df["keep_fixed"] = pd.Series(keep_fixed_bool_vector, index=new_df.
                                     index)

    output_df_list.append(new_df)
return output_df_list #a list of dataframes whose len==n_sample; each has
                       these columns: "distribution","
                       a_true","b_true","amplitude_true","
                       keep_fixed"

def param_randomizer(df, vary_only_amp = True):
    randomized_df = df.copy()
    for index, line in df.iterrows():
        dist_type = line["distribution"]
        params = np.asarray(line[["a_true","b_true","amplitude_true"]])
        if not line["keep_fixed"]: #must have an added a column with boolean
                                   values indicating to fix this
                                   particular function or not.

            #must make sure that amplitude is nonzero, and if dist_type=="
                                   maxwellian", must be non-
                                   zero

            if vary_only_amp:
                randomized_df.loc[index, "amplitude_true"] = np.random.
                                                                lognormal() *
                                                                randomized_df.loc[index,
                                                                "amplitude_true"]

            else:
                randomized_df.loc[index] = np.random.multivariate_normal(
                                                                params, get_covar_mat(
                                                                dist_type,params) )

    return randomized_df

def get_covar_mat(dist_type,params):#dist_type is a string
    df_dx_i_list = get_df_dx_i[dist_type](params)
    num_params = len(df_dx_i_list)
    #?unfinished
    return

def spectrum_generator(function_dataframe): #a 2D dataframe input
    function_list = []
    for index, line in function_dataframe.iterrows():
        dist_type = line["distribution"]
        params = line[["a_true","b_true","amplitude_true"]]
        function_list.append( function_pointers[dist_type]( *list(params) ) )
    return lambda x: sum([ f(x) for f in function_list ])

def return_AA1(params):
    return [params[2], params[2], 1]
def return_A1(params):
    return [params[1], 1]
def return_Watt_params(params):
    a,b,A = params #unpack list
    area = sqrt(pi/2)*A*sqrt(a**3 * b) * exp(a*b/4)
    return [area*(a+6)/(4*a) , area*(b+2)/(2*b), area*1/A]

```

```

get_df_dx_i = {
'normal' : return_AA1,
'normal_fixed_mean' : return_AA1,
'log_normal' : return_AA1,
'log_normal_fixed_mean' : return_AA1,
'maxwellian' : return_A1,
'maxwellian_fixed_mode' : return_A1,
'watt_spectrum' : return_Watt_params,
}

#parameterising functions that return lambda function objects
def normal_dist(*args):
    mu, sigma, amplitude = args[-3:]
    return lambda x: amplitude/sqrt(2*pi* sigma**2) * exp(-(x-mu)**2 / (2*sigma
**2) )

def lognormal_dist(*args):
    mu, sigma, amplitude = args[-3:]
    return lambda x: amplitude/(x*sigma*sqrt(2*pi)) * exp( -(log(x)-mu)**2 /(2*
sigma**2) )

def maxwellian_dist(*args):
    mode, amplitude = args[-2:]
    a = mode/sqrt(2)
    return lambda x: amplitude*sqrt(2/pi) * (x**2/a) * exp( -(x**2)/ (2 * (a**
2) ) )

def watt_spec(*args):
    a, b, amplitude = args[-3:]
    return lambda x: amplitude* exp( -x/a ) * np.sinh( sqrt(b * x) )

def save_numpy_array_with_comment_as_csv(fname, comment, array):
    comment = comment.split("\n")
    comment[0] = "#"+comment[0]
    comment[-1]= comment[-1]+"\\n"
    comment = "\\n#".join(comment)
    array = np.clip(array, 1, None)
    with open(fname,"a") as f:
        f.write(comment)
    with open(fname,"b+a") as f:
        np.savetxt(f,array,delimiter=",")
    return

def get_comment(for_spectra=True):
    if for_spectra:
        comment = ["Each row of this file list ONE spectrum",
"each column corresponds to the flux value of a specific an energy bin.",
" ",
"These will act as the labels with which the neural network will be
trained on /tested on."
]
    else: #otherwise this would be used to generate comments for csv files.
        comment = ["Each row of this file list the reaction rates obtained
after folding ONE spectrum",
"each column corresponds to the activities of a specific energy bin.",
"This will act as the features with which the neural network will be
trained on /tested on."
]
    return "\\n".join(comment)

function_pointers={ #dictionary that when called with the appropriate string,
acts as an alias to the function

```

```

'normal' : normal_dist,
'normal_fixed_mean' : normal_dist,
'log_normal' : lognormal_dist,
'log_normal_fixed_mean' : lognormal_dist,
'maxwellian' : maxwellian_dist,
'maxwellian_fixed_mode' : maxwellian_dist,
'watt_spectrum' : watt_spec,
}

if __name__=="__main__":
    #initialize parameters:
    Reactor_list = ["1_JAEA_FNS", "2_Frascati_NG", "3_ITER_DD", "4_ITER_DT", "
                    5_DEMO_HCPB_FW", "6_JET_FW", "
                    7_NIF_Ignition",
                    "8_IFMIF_DLi", "9_BWR_UO2_15", "10_BWR_MOX_15", "12_PWR_MOX_15", "13_Cf252", "
                    14_Maxwellian"]

    #<edit here>
    seed_val=0
    PLOT=False #Decide whether to show the plots or not
    target_gs = Reactor_list[0]
    save_file_prefixes = "../"+"GS_eq_"+target_gs
    save_file_prefixes += "_reference"
    n_sample = 300 #Choose number of feature:label pairs to be created
    keep_fixed_fraction = 1.0
    #choosing data source
    #</edit here>
    for Rxr in Reactor_list:
        method_list=["ACT", "TBMD", "VERDI"]
        # Rxr = Reactor_list[5]

        np.random.seed(seed_val)
        parameter_csv_suffix="_optimal_parameters.csv"
        df = pd.read_csv(Rxr+parameter_csv_suffix)
        response_matrix_suffix="_Response_Matrix.csv"

        #csv parameter's format is as follows:
        #for the case of normal distributions, a=mu, b=sigma; case of
        #maxwellian: b=mode;
        #unused parameters becomes 'nan' or 1

        #The true values to be plugged into various distributions are as
        #follows:

        df['a_true'] = df.a_fixed*df.a_corr
        df['b_true'] = df.b_fixed*df.b_corr
        df['amplitude_true'] = df.amplitude_fixed*df.amplitude_corr
        #except with the two cases where amplitudes were scaled logarithmically
        #using the correction factor:
        df.loc[df.distribution=="normal", "amplitude_true"] = df.
            amplitude_fixed * 10**(df.
            amplitude_corr-1)/sqrt(2*pi)
        df.loc[df.distribution=="normal_fixed_mean", "amplitude_true"] = df.
            amplitude_fixed * 10**(df.
            amplitude_corr-1)/sqrt(2*pi)
        df.loc[df.distribution=="log_normal", "amplitude_true"] = df.
            amplitude_fixed * 10**(2*(df.
            amplitude_corr-1))
        df.loc[df.distribution=="log_normal_fixed_mean", "amplitude_true"] = df.
            amplitude_fixed * 10**(2*(df.
            amplitude_corr-1))

```

```

#Obtain the group structure
gs = get_group_structure(target_gs)#get the flux values corresponding
                                     to the target group structure
groups_centroids = convert_to_centroid_values(gs)

#start reading and processing the function parameters
list_of_df = preprocess_df(df, n_sample=n_sample, keep_fixed_fraction=
                           keep_fixed_fraction) #
                           preprocess_df outputs a list of
                           dataframe with len=n_sample
randomized_list_of_df = [ param_randomizer(df_i) for df_i in list_of_df
                          ]
print("Randomized {0} dataframes of parameters for {1}".format(n_sample
                                                                , Rxr))

target_spectra = [ spectrum_generator(randomized_df)(groups_centroids)
                   for randomized_df in
                   randomized_list_of_df ] #
                   generate the features

file_structure_comment=get_comment()
save_numpy_array_with_comment_as_csv(save_file_prefixes+"_spectra.csv",
                                     file_structure_comment,
                                     target_spectra)

# np.savetxt(save_file_prefixes+"_spectra.csv",target_spectra,delimiter
            =",")
print("Finished generating {0} spectra for {1}, using the group
      structure of {2}".format(
n_sample,Rxr,target_gs))

response_matrix = {} #create dictionary to store the matrices
method_comment = get_comment()
for method in method_list:
    response_matrix[method] = np.genfromtxt(target_gs+"_"+method+
                                             response_matrix_suffix,
                                             delimiter=",")

    spectrum_file, reaction_rates_file=[], [] #spectrum file, reaction
                                             rate files
    reaction_rates = [ response_matrix[method].dot(spec) for spec in
                      target_spectra ] #fold to
                      get the labels
    save_numpy_array_with_comment_as_csv(save_file_prefixes+"_"+method+
                                         "_RR.csv", method_comment,
                                         reaction_rates)

    print("Folded each sample through the {0} system".format(method))

print("time taken in seconds =",time.time()-start_time)

```

## F Training and evaluating neural networks on the underdetermined simulation data

A demonstration of applying neuralnetworktrainer.py on the data generated by spectrum-randomizer.py .

script\_for\_demo.py

```

#!/home/ocean/anaconda3/bin/python3
from neuralnetworktrainer import *

```



```

inc="_including_folded_reaction_rates"
mse = "mean_squared_error"
mpse = "mean_pairwise_squared_error"

modification=[] #create empty list to store dictionaries, each specifying what
                 #modification to make to the default demo
                 #NN.

# modification.append(
#     {"session_name":"test", "loss_func":mse, "callbacks_applied":['
#                                     EarlyStopping'], "hidden_layer":[128,
#                                     256], "cutoff": 10})

modification.append(
    {"session_name": "_128_256_mse", "loss_func":mse, "callbacks_applied":['
        EarlyStopping'], "hidden_layer":[128
        , 256], "cutoff": 1800})

modification.append(
    {"session_name": "_256_256_mse", "loss_func":mpse, "callbacks_applied":['
        EarlyStopping'], "hidden_layer":[256
        , 256], "cutoff": 1800})

modification.append(
    {"session_name": "_128_256_mse_inc", "loss_func":mse+inc, "callbacks_applied
        ":['EarlyStopping'], "hidden_layer":
        [128, 256], "cutoff": 1800})

modification.append(
    {"session_name": "_256_256_mse_inc", "loss_func":mpse+inc, "
        callbacks_applied":['EarlyStopping']
        , "hidden_layer":[256, 256], "cutoff
        ": 1800})

if __name__=="__main__":
    for mod in modification:
        nn=NeuralNetworkHandler()
        nn.session_name=mod["session_name"]
        nn.hyperparameter["loss_func"] = mod["loss_func"]
        nn.hyperparameter["hidden_layer"]= mod["hidden_layer"]
        nn.callbacks_applied = mod["callbacks_applied"]
        nn.data_reordering_options["cutoff"] = mod["cutoff"]
        nn.read_demo_data()
        nn.trim_data()
        nn.shuffle()
        nn.split_into_sets()
        nn.preprocess_input()
        nn.build_model()
        nn.train_model()
        nn.auto_generate_session_name()
        nn.save_params_as_dictionary()
        nn.save_NN_weights()
        nn.plot_history()
        nn.plot_test_results_histogram()
        nn.compare_individual_spectra(silent_mode=True)
        # nn.plot_training_spectra(threshold=2)

```

## G Selecting from UKAEA and IAEA compendium

Rebinned spectra from the 212 IAEA + UKAEA compendium [37] were sorted into various training and testing sets using the following python program.

getrealdata.py

```
import numpy as np
```

```

import pandas as pd
TRAIN_SPLIT = 0.8
SHUFFLE_SEED= 0
'''
# will save one for each of the following
fusion
mcf
fission
commercial_fission
watt
high_energy
activations
every
'''

'''
This file collect all the spectra in generator into a single csv file,
#NORMALIZE THEM,
and then fold them all through the three activation system's response matrices
to get the activation rates.
'''

# Assume all *.txt files in this directory belongs to the spectrum.

def get_ACT_TBMD_VERDI_matrix(absolute_path):
    import os
    matrices = {} #store the three matrices in a dictionary.
    activation_system = ["ACT", "TBMD", "VERDI"]

    for system in activation_system:
        for file in os.listdir(absolute_path):
            if (system in file) and ("response_matrix" in file):
                labelled_matrix = pd.read_csv(absolute_path+file, header=None,
                                                index_col=0)
                matrices[system] = np.array(labelled_matrix) # add
                                                                reponse matrix to
                                                                dictionary
                # print(system, "has response matrix of shape", matrices[system]
                                                                ].shape)

    return matrices #return a dictionary storing the matrices as numpy array in
                                                                the values, corresponding to the
                                                                system name stored in the keys.

def normalize(one_dim_array):
    total = sum(one_dim_array)
    return one_dim_array/total, total

# set(list(spec_index["type"]))
# Want the numbers in PUL, so that the neuralnetworktrainer.run_real will use a
                                                                default of label_already_in_PUL=True
# Add the "ref_info" into neuralnetworktrainer.run_real as well.
# # save 1 metadata file + 1 spectra norm.csv file + 3 reaction rates for each
                                                                of the following:
'''
{'BT', # bombardment/Boron target
'CR', # cosmic ray
'HEA', # high energy activation
'IS', # instantaneous source (Americium)
'MA', # microtron activation

```

```

'PR', # Pressurized Reactor
'RFT', # reprocessing fuel technology
'UKAEA_FIS', # fission
'UKAEA_FUS', # fusion
'UKAEA_HEA',
'UKAEA_IS',
'UKAEA_PR'}
'''

spec_index = pd.read_csv("real_spectrum_index.txt", sep="\t")
types = spec_index["type"] # shorten the variable name
descriptions = spec_index["description"]

def get_matching_type(*strings):
    matching_loc = ( types=="") #get a list of all false
    for pattern in strings:
        if pattern.startswith("*"):
            pattern=pattern[1:]# remove the *
            matching_loc = np.logical_or (matching_loc, types.str.match("UKAEA_"
                                                                           "+pattern") )
            matching_loc = np.logical_or (matching_loc, types.str.match(pattern) )
            #add these matching patterns

    return matching_loc

def search_in_description(*strings):
    matching_loc = ( descriptions=="")
    for pattern in strings:
        matching_loc = np.logical_or (matching_loc, descriptions.str.contains(
                                                                           pattern) )

    return matching_loc

def get_rebinned_data(file_whole_path, gs):
    E, fluence =np.genfromtxt(file_whole_path).T
    assert all(E==gs[:-1]), "The energy group doesn't match the lower bound of
                               the reference group!"

    return fluence

def shuffle(truth_value_series): #reproducibly shuffle the dataframe
    #truth_value_series is a pd.Series object with one boolean value
    #corresponding to each row of the
    #dataframe, to represent whether or
    #not it's selected.

    np.random.seed(SHUFFLE_SEED)
    indices = list(truth_value_series[truth_value_series].index) #this extracts
    #the rows whose boolean values are "
    #True".

    np.random.shuffle(indices)
    return indices

if __name__=="__main__":
    spectra_classifications = {
        #fusion spectra
        "every" : shuffle(search_in_description("")),
        "fusion" : shuffle(get_matching_type("*FUS")), #19 of such spectra
        "mcf" : shuffle(search_in_description("-FW", "-VV", "ITER")), #13 of such
        #spectra

        #fission spectra
        "fission" : shuffle(get_matching_type("*FIS", "RFT", "BT", "*PR", "*IS")),
        #133 of such spectra
        "commercial_fission" : shuffle(get_matching_type("*FIS", "*PR")), #88 of

```

```

                                such spectra
"watt" : shuffle(get_matching_type("IS", "UKAEA-IS")), #watt spectra
                                without apparent moderating medium
#    5 of such spectra

#miscellaneous
"high_energy" : shuffle(get_matching_type("*HEA", "CR", "MA")), # spectra
                                containing a significant amount of
                                high energy particles
#    56 of such spectra
"activations" : shuffle(get_matching_type("*HEA", "MA", "RFT")), # spectra
                                of activated materials.
#    82 of such spectra

# assert sum(fusion + fission + high_energy + activations) == len(
                                spec_index), "Some doesn't belong to
                                any of the above categories!"
}
#For a few kinds of classifications of interest,
for specific_kind in ("every", "fusion", "fission"):
    sample_size = len(spectra_classifications[specific_kind])
    train_rows = round(TRAIN_SPLIT * sample_size)
    spectra_classifications[ specific_kind+"_train"] =
                                spectra_classifications[
                                specific_kind][:train_rows]
    spectra_classifications[ specific_kind+"_test" ] =
                                spectra_classifications[
                                specific_kind][train_rows:]

directory = "All_spectra_in_175/data_package_175convert/"

response_matrix = get_ACT_TBMD_VERDI_matrix(directory+ "../..")

gs = np.genfromtxt(directory + "175_gs.csv")
for selection_name, classification in spectra_classifications.items():
    #placeholder for the output dataframe.
    output_fluence = []
    output_rr = dict( [ (system,[]) for system in response_matrix.keys() ]
                      )
    normalization_constant_list = [] #placeholder for normalization
                                    constants to be added to the
                                    metadata dataframe.

    selected = spec_index.iloc[classification].copy()
    for f in selected["title"]: # read the strings from the title column
        raw_line = get_rebinned_data(directory+f+".txt", gs) #grab the
                                                                original line, and then
                                                                noramlize it.
        norm_line, norm_const = normalize(raw_line)
        #save the normalized output, normalization constant, and the 3
                                                                respective response rates.
        output_fluence.append( norm_line )
        for system, rr in output_rr.items():
            rr.append( response_matrix[system].dot( norm_line))
        normalization_constant_list.append( norm_const )

    # save the files outputted
    save_name = "real_"+selection_name+"_normed"
    pd.DataFrame(output_fluence).to_csv(save_name+".csv", header=False,
                                        index=False)

```

```

selected["normalization_constant"] = normalization_constant_list
selected.to_csv(save_name+"_ref_info.csv", header=True, index=False)

for system, rr in output_rr.items():
    df = pd.DataFrame(rr)
    df.to_csv(save_name+"_"+system+".csv", header=False, index=False)

```

## H hyperparameter input controller

Input files for hyperparametertrainer.py using the following code, by iterating through a list of hyperparameters of interest, thus effectively performing a grid search over all hyperparameters.

hyperparameterinput.py

```

import numpy as np
# from matplotlib import pyplot as plt
import pandas as pd
from itertools import product
import hashlib
import sys
import glob
import os
import shutil

def generate(*filename):
    sheet = pd.DataFrame([], columns=["loss_func", "hidden_layer", "
                                     learning_rate", "num_epochs", "files
                                     ", "session_name",
                                     "train_loss", "train_mae", "train_mse",
                                     "val_loss" , "val_mae" , "val_mse" ,
                                     "test_loss", "test_mae", "test_mse",
                                     "std_CE_rr", "optimal_epoch"])

    #loss functions
    loss_func_list = ["mean_squared_error",
                      # "cosine_distance",
                      "mean_pairwise_squared_error",
                      "mean_squared_error_including_folded_reaction_rates",
                      "
                      mean_pairwise_squared_error
                      "]

    #hidden layers
    hidden_layer_list=[]
    ,,,
    #discarded choices of hidden_layers as listed as follows:
    hidden_layer_list.append([])
    hidden_layer_list.append([256])
    hidden_layer_list.append([128, 256])
    hidden_layer_list.append([64, 128, 256])
    for n in range(1,6):
        hidden_layer_list.append([256,]*n)
    ,,,
    for n in range(6):
        increasing_node_list = np.logspace(5,8, n ,base=2).astype(int)
        hidden_layer_list.append( list(increasing_node_list) )

```

```

#learning rate
learning_rate_list = np.logspace( -2,-9, 43)
# learning_rate_list = list(np.logspace(-6, -9, 10))

#training and testing set.
files_list = [ #self verifying
    ("every", "every"),
    ("fusion", "fusion"),
    ("fission", "fission"),
    #cross verifying
    ("fission", "fusion"),
    ("fusion", "fission"),
    #generalization within each category
    ("mcf", "fusion"),
    ("commercial_fission", "fission"),
    #cross verifying
    ("activations", "high_energy"),
    ("high_energy", "activations"),
    #the fission spectra should already contain enough information
    #to deduce the watt
    #spectrum
    ("fission","watt"),
    #for fun, see if the fusion spectra contain enough information
    #to deduce the watt
    #spectrum
    ("fusion", "watt"),]

p = product(loss_func_list, hidden_layer_list, learning_rate_list,
            files_list)

while True:
    try:
        loss_func, hidden_layer, learning_rate, files = next(p)
    except StopIteration:
        break

    #hash out a name:
    line = str(loss_func) + str(hidden_layer) + str(learning_rate) + str(
        files)
    name = hashlib.shake_256( line.encode("utf-8") ).hexdigest(6)

    # num_epochs = int( np.clip(10** ( round(len(hidden_layer))-1 ) * round(
        1/learning_rate), 10, 1E5) ) #
        limit the number of epoch to
        100000.

    num_epochs = 10000
    #add these data into the end of the spreadsheet
    sheet.loc[ len(sheet.index) ] = [ loss_func, hidden_layer,
        learning_rate, num_epochs, files
        , name, 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 10000 ] #will
        leave the loss columns empty.

    assert len(set(sheet["session_name"]))==len(sheet.index), "there are
        repetitions of the hashed names; try
        using a longer hexdigest size."

    print("number of rows saved =", len(sheet.index) )
    if len( filename)==0:
        fname = "hyperparameterlist.csv"
    else:
        fname = filename[0]

```

```

sheet.to_csv(fname, index=False)
print("saved as", fname)

def _write_one_dict_with_EarlyStopping(f, row):
    f.write("\n{\n")
    f.write('response_matrix_file : "response_matrix_ACT_175_gs.csv"' + ",\n")
    f.write('group_structure_file : "175_gs.csv"' + ",\n")
    f.write("loss_func : " + "'" + str(row["loss_func"]) + "'" + ",\n")
    f.write("hidden_layer : " + str(row["hidden_layer"]) + ",\n")
    f.write("learning_rate : " + str(row["learning_rate"]) + ",\n")
    f.write("num_epochs : " + str(row["num_epochs"]) + ",\n")
    train, test = [ i.replace("(", "").replace(")", "").strip("'") for i in row["files"].split(", ") ]

    if train==test:
        train=train+"_train"
        test=test+"_test"
    f.write('train_feature_file: "real_'+train+'_normed_ACT.csv"' + ",\n")
    f.write('train_label_file : "real_'+train+'_normed.csv"' + ",\n")
    f.write('test_feature_file : "real_'+test+'_normed_ACT.csv"' + ",\n")
    f.write('test_label_file : "real_'+test+'_normed.csv"' + ",\n")
    f.write('ref_info_file : "real_'+test+'_normed_ref_info.csv"' + ",\n")
    f.write('session_name : "' + str(row["session_name"]) + "'" + ",\n")
    f.write("callbacks_applied : ['EarlyStopping'] ,") #this callback by
                                                    default restores the best weight.

    f.write("}\n")

def append_dict(*filename):
    if len (filename)==0:
        fname = "hyperparameterlist.csv"
    else:
        fname = filename[0]
    sheet = pd.read_csv(fname, index_col=None)
    with open("real_hyperparameter_tweaking.txt", "a") as f:
        for _, row in sheet.iterrows():
            _write_one_dict_with_EarlyStopping(f,row)

def split_dict(num_jobs, *filename):
    assert num_jobs<=999, "current filename syntax restricts the number of jobs
                           to 3 digits"

    if len (filename)==0:
        fname = "hyperparameterlist.csv"
    else:
        fname = filename[0]
    print("reading from {1}, splitting into {0} dictionaries".format(num_jobs,
                                                                    fname))
    sheet = pd.read_csv(fname, index_col=None)
    num_rows = len(sheet.index)

    rows_per_file = int(np.ceil(num_rows/num_jobs))
    for n in range(num_jobs):
        with open("job_number_"+str(n).zfill(3)+".txt", "w") as f:
            for _, row in sheet.iloc[ rows_per_file*n : rows_per_file*(n+1) ].
                iterrows():
                    _write_one_dict_with_EarlyStopping(f,row)

def search_in_df(*args):
    sheet = pd.read_csv("hyperparameterlist.csv", index_col=None)
    with open("hyperparameterlist.csv") as f:
        lines = f.readlines()[1:] #ignore the header line

```

```

mask = [True,]*len(sheet.index)
for arg in args:
    new_mask = [ (arg in line) for line in lines ]
    print(sum(new_mask), "matches for ", arg)
    mask = np.logical_and(mask, new_mask)

if sum(mask)==0:
    print("No matching results!")
    return
elif sum(mask)>1:
    print("Multiple lines are found to match. the first five are as follows
          :")

print(sheet[mask].head())
return

if __name__=="__main__":
    print("This version of hyperparameterinput.py applies EarlyStopping to
          prevent overfitting.")

    try:
        arg = sys.argv[1]
    except IndexError:
        print("type one of the following words after the program name:")
        print("generate")
        print("write")
        print("split")
        print("search")
        exit()

    if arg=="generate":
        generate(*sys.argv[2:])
    elif arg=="write":
        append_dict(*sys.argv[2:])
    elif arg=="split":
        if len(sys.argv)==2:
            split_dict(132) #by default split into 132 dictionaries
        else:
            split_dict(int(sys.argv[2]), *sys.argv[3:])
    elif arg=="search":
        search_in_df(*sys.argv[2:])

```

This program can be used to split the into multiple jobs, which can then be submitted to a cluster, parallellizing the process and massively reducing the training and evaluation time of the neural networks. This is done by calling the program with `python hyperparameterinput.py split`

## I hyperparameter optimization searching

List the hyperparameter, training- and testing-sets used to evaluate the neural network on, when the hash\_name of the neural network is given.

hyperparameteroutput.py

```

import numpy as np
# from matplotlib import pyplot as plt
import pandas as pd
from itertools import product
import sys
import glob
import os
import shutil

```



```

def search_in_df(*args):
    verbose=False
    sorting=False
    sheet = pd.read_csv("hyperparameterlist.csv", index_col=None)
    with open("hyperparameterlist.csv") as f:
        lines = f.readlines()[1:] #ignore the header line

    mask = [True,]*len(sheet.index)
    for arg in args:
        if arg=="-v":
            print("Setting verbose to True")
            verbose=True
        elif arg=="-s":
            print("Sorting the outputted dataframe according to the last
                    argument provided={}".format(
                        args[-1]))

            sorting=True
        else:
            if not sorting:
                new_mask = [ (arg in line) for line in lines ]
                print(sum(new_mask), "matches for ", arg)
                mask = np.logical_and(mask, new_mask)

    if sum(mask)==0:
        print("No matching results!")
        return
    elif sum(mask)>1:
        print("{0} lines are found to match. the first five are as follows:".
                format(sum(mask)))

    region_of_interest = sheet[mask]
    if sorting:
        region_of_interest=region_of_interest.sort_values(by=[arg])
    if verbose:
        print(region_of_interest)
        print("with the name(s)")
        print(region_of_interest["session_name"])
    else:
        print(region_of_interest.head())
        print("with the name(s)")
        print(region_of_interest["session_name"].head())
    return

def fill_in_loss_values():
    sheet = pd.read_csv("hyperparameterlist.csv", index_col=None)
    #try to find the hash in the filename

    for ind, row in sheet.iterrows():
        name = row["session_name"]
        matching_txt = glob.glob(name+"_params.txt")
        if len(matching_txt)==0:
            print("Params file for neural network with hash='{0}' is not found/
                    not generated yet.".format(
                        name), end='\r', flush=True)

            continue
        elif len(matching_txt)>1:
            print("Warning: multiple params*.txt of hash={0} is found!".format(
                    name))

            [ print(i) for i in matching_txt]
            print("Using the loss value in the last one.")

```

```

with open(matching_txt[-1]) as f:
    lines = f.readlines()
def find_in_file(word):
    error_message_line = [line.strip() for line in lines if line.
                           startswith("session_name :")
                           ]

    loss_lines = [ line.strip() for line in lines if line.startswith(
                           word+" :")] #choose the
                                           matching line

    if len(loss_lines)!=1:
        print("\nNumber of matching lines found =" +str(len(loss_lines))
              + " !")

        if word=="std_of_log_of_C_over_E_reaction_rates": #only let it
                                                           slip if it's because the
                                                           folding process messed
                                                           up and created a
                                                           negative value.

            print("      Ignoring the missing C/E value for line "+
                  error_message_line[0]
                  ])

            print("      continuing 'fill' action")
            print("      |")
            print("      |")
            print("      |")
            print("      |")
            print("      |")
            return
        else:
            exit()

    loss_value = float(loss_lines[0].split(":")[1].strip().strip(",") )
                           #take the part after the
                           ':', and remove the '\n' and
                           ','

    if not np.isfinite(loss_value):
        print("\n"+error_message_line[0]+"has a non-finite value of {0}
              = {1}".format(word, str(
              loss_value)) )

    return loss_value

#below is an extremely inefficient way of filling in the loss values.
sheet.at[ind,"train_loss"]=find_in_file("loss")
sheet.at[ind,"train_mae"]=find_in_file("mean_absolute_error")
sheet.at[ind,"train_mse"]=find_in_file("mean_squared_error")
sheet.at[ind,"val_loss"] =find_in_file("val_loss")
sheet.at[ind,"val_mae"]  =find_in_file("val_mean_absolute_error")
sheet.at[ind,"val_mse"]  =find_in_file("val_mean_squared_error")
sheet.at[ind,"test_loss"]=find_in_file("test_loss")
sheet.at[ind,"test_mae"] =find_in_file("test_mean_absolute_error")
sheet.at[ind,"test_mse"] =find_in_file("test_mean_squared_error")
sheet.at[ind,"std_CE_rr"]=find_in_file("
                                std_of_log_of_C_over_E_reaction_rates
                                ")

sheet.at[ind,"num_epochs"]=int(find_in_file("num_epochs"))
sheet.to_csv("hyperparameterlist.csv", index=False) #overwrite the old file
.

def copy(source_list, dest):
    if len(source_list)!=1:
        assert len(source_list)>0, "no matching files found!"
        print("multiple matching files found, they are listed below. Using the
              last one... \n{0}".format("\n".

```

```

join(source_list)))

shutil.copy(source_list[-1], dest)

def rearrange(*cols): #rearrange folder structure according to the column name
    sheet = pd.read_csv("hyperparameterlist.csv", index_col=None)
    for col in cols:
        assert col in sheet.columns, "column {0} not found!".format(col)
    top_level_name = "sort_by_+"+"-".join([str(col) for col in cols])

    sets = [list(set(sheet[col])) for col in cols]
    #converting the above sets into folder names
    folder_name = []
    for s in sets:
        new_row = []
        for i in s:
            i=i.replace("'", "").strip("[]").strip "()".replace(", ", "-")
            if i == "": i="empty"
            new_row.append(i)
        folder_name.append(new_row)
    # folder_name = [ [i.replace("'", "").strip("[]").strip "()".replace(", ", "-") for i in s] for s in sets] #
    # turn each item in the set into a folder name
    # folder_name = [ [elem for elem in row if elem!=" " else "empty"] for row
    # in folder_name]

    #use for loop and the itertools.product function to create all
    #subdirectories

    path_name = [ [ top_level_name, ], ]
    for i in range(len(cols)):
        next_level_names = [ os.path.join(*pair) for pair in product(path_name[-1], folder_name[i]) ]

        path_name.append(next_level_names)
        for folder in path_name[-1]:
            try:
                os.mkdir(folder)
            except FileExistsError:
                pass

    #select the matching hashed names and pull them into the correct folder
    j = 0
    for matching_criteria in product(*sets):
        mask = [True,]*len(sheet.index)
        for i in range(len(matching_criteria)):
            mask = np.logical_and(mask, sheet[cols[i]]==matching_criteria[i])
        matching_names = sheet[mask]["session_name"]
        folder = path_name[-1][j]
        folder_errorvar = os.path.join(folder, "errorvar")
        folder_deviationdistr = os.path.join(folder, "deviationdistr")
        try:
            os.mkdir(folder)
            os.mkdir(folder_errorvar)
            os.mkdir(folder_deviationdistr)
        except FileExistsError:
            pass
        for name in matching_names:
            matching_txt = glob.glob("*"+name+"_params.txt")
            copy(matching_txt, folder)
            matching_errorvar = glob.glob("lossabove1e*/errorvar/*"+name+"*.png")

```

```

        copy(matching_errorvar, folder_errorvar)
        matching_deviationdistr = glob.glob("lossabove1e*/deviationdistr/*"
                                             +name+".png")
        copy(matching_deviationdistr, folder_deviationdistr)
    j+=1
    # assert j==len(next_level_names), "at this point j should equal to the
                                     number of lowest level files"

if __name__=="__main__":
    try:
        arg = sys.argv[1]
    except IndexError:
        print("type one of the following words after the program name:")
        print("fill")
        print("search")
        print("sort")
        exit()

    if arg=="fill":
        fill_in_loss_values()
    elif arg=="search":
        search_in_df(*sys.argv[2:])
    elif arg=="sort":
        rearrange(*sys.argv[2:])

```

## J Loss value visualizer

When given the names of the training- and testing-set, the following code show the loss values (and other metrics) of the neural networks with different hyperparameters achieved on them. This is plotted as a heat map, over the two dimensions of hyperparameters varied, which are ‘number of layers’ (y-axis) and ‘learning rate’ (x-axis) respectively.

hyperparameteroptimizer.py

```

import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import sys
import glob
import os
import numpy as np

def plot3d(*args):
    sheet = pd.read_csv("hyperparameterlist.csv", index_col=None)
    file_list = list(set(sheet["files"]))
    folder_list = [ i.replace("'", "").strip("[]").strip "()".replace(", ", "-")
                    for i in file_list ] # get it as
                                         the prettier names.

    matching_file_pairs = []
    for arg in args:
        for i in range(len(folder_list)):
            if folder_list[i].startswith(arg):
                matching_file_pairs.append(file_list[i])
    for train_test in matching_file_pairs:
        raw_data = sheet[sheet["files"]==train_test].drop(columns=["files", "
                                                                num_epochs", "session_name"])
    set_of_loss_func = ['mean_squared_error',
                        'mean_squared_error_including_folded_reaction_rates',

```

```

        'mean_pairwise_squared_error',
        ,
        mean_pairwise_squared_err
        ',']

for loss_func_name in set_of_loss_func:
    print("showing plots of loss_func="+loss_func_name)
    show_each_metric(raw_data[raw_data["loss_func"]==loss_func_name],
                    train_test, loss_func_name)

def pivot_and_plot_heatmap(df,metric):
    pivot_table = df.pivot(index="hidden_layer", columns="learning_rate",
                            values=metric)
    pivot_table.columns=np.array2string(pivot_table.columns, precision=2).strip
    ('[]').split() #bodged together in a
                    hurry.
    pivot_table = pivot_table.reindex([ #reorder the pivot table rows so that
                                        it goes ascending.
                                        '[32, 53, 90, 152, 256]',
                                        '[32, 64, 128, 256]',
                                        '[32, 90, 256]',
                                        '[32, 256]',
                                        '[32]',
                                        '[]'
                                        ])
    pivot_table = np.log10(pivot_table) #taking log10 to normalize the loss-
                                        values.
    handle= sns.heatmap(pivot_table, annot=True)
    handle.set_xticklabels(handle.get_xticklabels(), rotation=-15)
    handle.set_yticklabels(handle.get_yticklabels(), rotation=-75)
    return handle

def show_each_metric(result_of_training_on_one_loss_func, train_test,
                    loss_func_name):
    metrics_that_i_care_about = ['val_loss', 'val_mse', 'test_loss', 'test_mse',
                                , 'std_CE_rr']
    for metric in result_of_training_on_one_loss_func.columns[-10:]:
        if metric in metrics_that_i_care_about:
            pivot_and_plot_heatmap(result_of_training_on_one_loss_func, metric)
            plt.title("log of "+metric+" of "+train_test+"\n optimized on "+
                    loss_func_name)

            plt.show()
if __name__=="__main__":
    plot3d(*sys.argv[1:])

```

## K Parametrisation of the FISPACT reference spectra

distribution	$\mu$	$\sigma$	A	$\mu_{corr}$	$\sigma_{corr}$	$A_{corr}$
log-normal	-1.48e+01	1.20e+00	1.00e+00	1.0	0.6536070787201796	0.6465171468399362
log-normal	-1.17e+01	1.20e+00	5.54e+01	1.0	0.6923014193474084	0.41834840464375106
log-normal	-8.61e+00	1.20e+00	3.07e+03	1.0	0.5802541895436414	0.3391941243798774
log-normal	-5.52e+00	1.20e+00	1.70e+05	1.0	0.769982827091882	0.5451675762326162
log-normal	-2.43e+00	1.20e+00	9.40e+06	1.0	0.6705439760036781	0.6056530392573959
log-normal	6.55e-01	1.20e+00	5.20e+08	1.0	0.6485812808091918	0.6213932412543168
log-normal	3.74e+00	1.20e+00	2.88e+10	1.0	0.33687762989691133	1.754564218248171
log-normal	-1.44e+01	1.50e+00	1.00e+09	1.0	1.1087020488776023	1.2837241483881578
log-normal	-8.60e+00	1.50e+00	3.22e+11	1.0	0.7012835343191862	0.6774642293787084
log-normal	-2.83e+00	1.50e+00	1.04e+14	1.0	0.9732542967472964	1.0011605034609532
log-normal	2.94e+00	1.50e+00	3.33e+16	1.0	0.9872173030484698	0.99379684387792
normal	1.41e+01	4.00e-01	1.00e+19	1.023	1.0500657332932959	1.128662095083972
log-normal	-1.54e+01	1.10e+00	1.00e+03	1.0	1.0295086712444834	1.0871973190897086
log-normal	-1.18e+01	1.10e+00	3.59e+04	1.0	1.0470576309590907	1.1133327728488918
log-normal	-8.26e+00	1.10e+00	1.29e+06	1.0	0.9512570373593815	1.0428424889188541
log-normal	-4.67e+00	1.10e+00	4.64e+07	1.0	1.1561895212201019	1.1532829852822521
log-normal	-1.09e+00	1.10e+00	1.67e+09	1.0	1.0181223850843093	1.018626898626218
normal	1.41e+01	4.00e-01	1.00e+10	1.0	1.32722437503459	1.8671520930196117
normal	2.45e+00	1.00e-01	1.00e+12	1.0	0.8152984470618088	0.5530754449242801
log-normal	-1.26e+01	2.00e+00	1.00e+05	1.0	1.094149298940297	1.17454890924765
log-normal	-7.12e+00	2.00e+00	2.47e+07	1.0	1.7120265659905378	1.12525321665773
log-normal	-1.61e+00	2.00e+00	6.08e+09	1.0	1.2209065015914118	1.5577920026942778
log-normal	3.89e+00	2.00e+00	1.50e+12	1.0	1.0538009149767102	1.5525710802981993
normal	1.41e+01	4.00e-01	1.00e+14	1.0	0.9946385566258605	1.1958058458498946
log-normal	3.00e+00	2.00e+00	1.00e+13	1.0	0.985831182477408	1.191099325784018
log-normal	-3.20e+00	2.00e+00	1.00e+11	1.0	1.0163826189572225	1.037504496747917
normal	1.41e+01	4.00e-01	1.00e+14	1.0	1.132214159680078	1.3221159758391146
log-normal	-5.60e+00	2.30e+00	4.00e+05	1.0	0.912896343655827	0.9707283164160951
log-normal	-4.80e+00	5.00e-01	1.00e+06	1.0	0.9724734344690737	1.3093741751167056
log-normal	3.00e+00	1.80e+00	5.00e+09	1.0	1.1626572027366295	0.8373153796607655
normal	1.41e+01	4.00e-01	1.00e+10	1.0	1.1868044101095476	1.555176012813058
normal	1.35e+01	2.00e+00	2.00e+20	1.0	1.0094598165344801	1.0398929376955228
log-normal	-2.40e+00	3.50e-01	1.00e+14	1.0	0.9999755394847119	1.0437028488962274
log-normal	-1.43e+00	3.50e-01	5.54e+14	1.0	1.0040524756949494	1.2576232255047286
log-normal	-4.47e-01	3.50e-01	3.07e+15	1.0	1.0213050801706227	1.2668031933646988
log-normal	5.31e-01	3.50e-01	1.70e+16	1.0	1.0185780833110203	1.3543132306863206
log-normal	1.51e+00	3.50e-01	9.40e+16	1.0	1.0622261078184665	1.1481233202529897

Table 5: In descending order: each section represents the parameters used to parametrise the spectra of: JAEA-FNS, Frascati-NG, ITER-DD, ITER-DT, DEMO-HCPB-FW, JET-FW, NIF-Ignition. The 2-4<sup>th</sup> columns indicate the guess value inputted, while the last 3 columns indicate the correction factor multiplied onto them. E.g.  $\mu_{final} = \mu * \mu_{corr}$

# L neutron spectra extraed from the IAEA + UKAEA compendium

List of all fission data

[H] title	type	Measured/Calculated
PR_PWR_CZECH_6	PR	Czech PWR, circ. pumps, near cold side, p2
PR_BWR_DUNG_1	PR	BWR, Dungeness, boiler cell
RFT_PU_3	RFT	Pu reprocessing plant, fuel pin assembly
RFT_MOX_4	RFT	Fresh MOX, borated water shield
PR_GCR_1	PR	CH1 GC reactor
RFT_PU_2	RFT	Pu reprocessing plant, well shielded
PR_PWR_CZECH_2	PR	Czech PWR, check room, under reactor, p5
PR_BWR_SW_4	PR	BWR (Switzerland), under access
PR_GCR_3	PR	Trawsfynydd GC reactor, position S3
PR_PWR_CZECH_4	PR	Czech PWR, check room, under reactor, p13
PR_BWR_SW_13	PR	PWR (Switzerland), behind generator
UKAEA_029_EBR-2	UKAEA_PR	EBR-2 spectra in 29 energy groups
PR_BWR_SW_8	PR	BWR (Switzerland), near lock, closed
PR_PWR_CZECH_7	PR	Czech PWR, circ. pumps, hot side, p3
PR_BWR_SW_16	PR	PWR (Switzerland), 33 cm behind door
UKAEA_1102_BWR-MOX-Gd-0	UKAEA_PR	BWR-MOX-Gd-0 spectra in 1102 energy groups
UKAEA_100_HFIR-lowres	UKAEA_PR	HFIR-lowres spectra in 100 energy groups
UKAEA_1102_PWR-MOX-15	UKAEA_PR	PWR-MOX-15 spectra in 1102 energy groups
BT_FRM_2	BT	FRM II beam, unfiltered
RFT_PU_1	RFT	Pu reprocessing plant, little shielding, location 1
PR_BWR_SW_14	PR	PWR (Switzerland), at reactor axis
PR_PWR_WOLFCREEK_3	PR	PWR, Wolf Creek, power 50%, 2026 level by valves
RFT_PU_USA_4	RFT	TRU plant, at conduit exit, less shielding
PR_PWR_CZECH_13	PR	Czech PWR, reactor hall, near cap, p10
IS_TRU_6	IS	TRU plant, 25 g AmO <sub>2</sub> in container
PR_BWR_SW_2	PR	BWR (Switzerland), at bend of maze
PR_BWR_CAORSO_8	PR	BWR, Caorso, position 4
PR_PWR_WOLFCREEK_1	PR	PWR, Wolf Creek, power 50%, at PH 2047 level
UKAEA_1102_PWR-UO2-Gd-15	UKAEA_PR	PWR-UO <sub>2</sub> -Gd-15 spectra in 1102 energy groups
PR_BWR_DUNG_3	PR	BWR, Dungeness, on the walkway
PR_BWR_CAORSO_3	PR	BWR, Caorso, position 3
PR_BWR_SW_5	PR	BWR (Switzerland), at stairwell
BT_LVR15_1	BT	LVR-15, epithermal beam
UKAEA_1102_BWR-MOX-Gd-15	UKAEA_PR	BWR-MOX-Gd-15 spectra in 1102 energy groups
PR_PWR_CZECH_10	PR	Czech PWR, reactor hall, at cap, p7
IS_TRU_9	IS	25 Ci Am ceramics, no shield
RFT_WWER_CASK_3	RFT	Transport cask C30/KB54, in corridor at 45 cm
RFT_WWER_CASK_1	RFT	Transport cask C30/KB54, in a hall at 45 cm
PR_GCR_2	PR	CH2 GC reactor
BT_BMRR_1	BT	BMRR spectra, filtered by 34 cm Al/AlF <sub>3</sub>
UKAEA_172_Phenix	UKAEA_PR	Phenix spectra in 172 energy groups
PR_TRAWS_1	PR	Trawsfynydd, filter gallery
RFT_TN12_1	RFT	TN-12 fuel container at Valognes
RFT_PU_USA_1	RFT	TRU plant, lightly shielded glovebox 1
RFT_VALDUC_1	RFT	Pu reprocessing at Valduc
RFT_PU_USA_2	RFT	TRU plant, heavily shielded glovebox 1
PR_HINKLEY_2	PR	Hinkley Point, filter gallery
BT_ACCEPL_1	BT	Accelerator based epithermal beam

PR_RINGHALS_3	PR	Ringhals, point P
UKAEA_616_HFR-high	UKAEA_PR	HFR-high spectra in 616 energy groups
PR_BWR_SW_15	PR	PWR (Switzerland), 40 cm behind door
PR_BWR_CAORSO_4	PR	BWR, Caorso, position 4
PR_BWR_SW_6	PR	BWR (Switzerland), 16 m level, near pump
BT_BMRR_1_1	BT	BMRR beam
PR_BWR_SW_10	PR	PWR (Switzerland), in front of containment
PR_BWR_CAORSO_7	PR	BWR, Caorso, position 3
RFT_MOX_2	RFT	MOX transport cask, ver. 03
UKAEA_198_PWR-RPV	UKAEA_PR	PWR-RPV spectra in 198 energy groups
UKAEA_407_Bigten	UKAEA_FIS	Bigten spectra in 407 energy groups
UKAEA_1102_BWR-UO2-Gd-40	UKAEA_PR	BWR-UO2-Gd-40 spectra in 1102 energy groups
BT_BMRR_2	BT	BMRR spectra, filtered by 22 cm <sup>7</sup> LiF
UKAEA_1102_PWR-UO2-15	UKAEA_PR	PWR-UO2-15 spectra in 1102 energy groups
BT_BMRR_3	BT	BMRR spectra, filtered by 17 cm D <sub>2</sub> O
RFT_1393_2	RFT	Transport cask 1393(2)
BT_BNCT_PETTEN_1	BT	BNCT in Petten, HB11 filtered beam
BT_FRM_3	BT	FRM II beam, filtered
UKAEA_1102_BWR-UO2-Gd-15	UKAEA_PR	BWR-UO2-Gd-15 spectra in 1102 energy groups
PR_PWR_GOSGEN_1	PR	PWR, Gosgen, position 1
UKAEA_198_BWR-RPV	UKAEA_PR	BWR-RPV spectra in 198 energy groups
RFT_PU_USA_5	RFT	TRU plant, lightly shielded glovebox 2
PR_RINGHALS_4	PR	Ringhals, point B2
IS_TRU_7	IS	TRU plant, 244Cm in glovebox, no shield
PR_BWR_CAORSO_2	PR	BWR, Caorso, position 2
RFT_PU_USA_3	RFT	TRU plant, at operator desk
UKAEA_1102_PWR-UO2-0	UKAEA_PR	PWR-UO2-0 spectra in 1102 energy groups
PR_RINGHALS_5	PR	Ringhals, point B4
PR_BWR_CAORSO_5	PR	BWR, Caorso, position 1
PR_PWR_WOLFCREEK_2	PR	PWR, Wolf Creek, power 50%, 2 m from PH 2047 level
PR_RINGHALS_1	PR	Ringhals, point D
PR_BWR_CAORSO_6	PR	BWR, Caorso, position 2
PR_BWR_SW_7	PR	BWR (Switzerland), 16 m level, near tap
UKAEA_172_Paluel	UKAEA_PR	Paluel spectra in 172 energy groups
RFT_POLLUX_2	RFT	Pollux container above ground level
PR_BWR_CAORSO_1	PR	BWR, Caorso, position 1
PR_PWR_CP_1	PR	CP reactor, site 5, SPUNIT code
PR_PWR_CZECH_8	PR	Czech PWR, circ. pumps, near door, p4
PR_BWR_SW_9	PR	BWR (Switzerland), near lock, open
RFT_MOX_1	RFT	MOX transport cask, ver. 02
PR_PWR_CP_2	PR	CP reactor, site 6, SPUNIT code
PR_PWR_CZECH_3	PR	Czech PWR, check room, in middle, p12
RFT_WWER_CASK_4	RFT	Transport cask C30/KB54, in corridor at 2 m
PR_PWR_CZECH_11	PR	Czech PWR, reactor hall, at valve, p8
IS_TRU_8	IS	AmBe sources in gloveboxes in line
PR_BWR_SW_11	PR	PWR (Switzerland), 1 m from platform
UKAEA_070_Cf252	UKAEA_IS	Cf252 spectra in 70 energy groups
UKAEA_1102_PWR-MOX-40	UKAEA_PR	PWR-MOX-40 spectra in 1102 energy groups
PR_PWR_CZECH_12	PR	Czech PWR, reactor hall, at generator, p9
UKAEA_1102_PWR-MOX-0	UKAEA_PR	PWR-MOX-0 spectra in 1102 energy groups
PR_HINKLEY_1	PR	Hinkley Point, pile cap
BT_ACC_2	BT	Gantry spectrum
PR_PWR_WOLFCREEK_4	PR	PWR, Wolf Creek, power 100%, at PH 2047 level



PR_PWR_CZECH_9	PR	"Czech PWR, circ. pumps, 4 & 4, p11 "
PR_BWR_SW_3	PR	BWR (Switzerland), at drywell
UKAEA_1102.BWR-UO2-Gd-0	UKAEA_PR	BWR-UO2-Gd-0 spectra in 1102 energy groups
BT_ACC_1	BT	Accelerator based spectrum, 2.5 MeV protons on 7Li
RFT_POLLUX_1	RFT	Pollux container in salt mine
RFT_LK100_1	RFT	LK-100 fuel container at La Hague, position1
UKAEA_616.HFR-low	UKAEA_PR	HFR-low spectra in 616 energy groups
RFT_MOX_5	RFT	Fresh MOX, no shield
RFT_NTL_1	RFT	Transport cask NTL-111, at 115 cm
RFT_HANAU_1	RFT	Fission material deposition, BS measurements
PR_PWR_WOLFCREEK_5	PR	PWR, Wolf Creek, power 100%, 2026 level at loop penetrati
UKAEA_172.Superphenix	UKAEA_PR	Superphenix spectra in 172 energy groups
RFT_WWER_CASK_2	RFT	Transport cask C30/KB54, in a hall at 2 m
RFT_MOX_3	RFT	New MOX at 20 cm, no shield
PR_PWR_CZECH_5	PR	Czech PWR, check room, corridor, p6
PR_PWR_CP_4	PR	CP reactor, site 4, YOGI code
RFT_HANAU_2	RFT	Fission material deposition, LS measurements
UKAEA_1102.PWR-UO2-40	UKAEA_PR	PWR-UO2-40 spectra in 1102 energy groups
PR_GCR_4	PR	Trawsfynydd GC reactor, position S4
UKAEA_1102.PWR-UO2-Gd-40	UKAEA_PR	PWR-UO2-Gd-40 spectra in 1102 energy groups
PR_RINGHALS_2	PR	Ringhals, point E
BT_SPALL_1	BT	Spallation source, 72 MeV protons on W
RFT_LK100_2	RFT	LK-100 fuel container at La Hague, position 2
PR_BWR_SW_12	PR	PWR (Switzerland), 3 m from platform
PR_PWR_GOSGEN_2	PR	PWR, Gosgen, position 2
UKAEA_1102.PWR-UO2-Gd-0	UKAEA_PR	PWR-UO2-Gd-0 spectra in 1102 energy groups
RFT_PU_5	RFT	Pu reprocessing plant, little shielding, location 5
RFT_NTL_2	RFT	Transport cask NTL-111, at 367 cm
RFT_PU_4	RFT	Pu reprocessing plant, little shielding, location 4
PR_PWR_CP_3	PR	CP reactor, site 4, SPUNIT code
PR_PWR_CZECH_1	PR	Czech PWR, circ. pumps, cold side, p1
PR_BWR_DUNG_2	PR	BWR, Dungeness, on the roof
BT_FRM_1	BT	FRM I beam
RFT_1392_1	RFT	Transport cask 1392(1)
UKAEA_1102.BWR-MOX-Gd-40	UKAEA_PR	BWR-MOX-Gd-40 spectra in 1102 energy groups
PR_BWR_SW_1	PR	BWR (Switzerland), at maze entrance

List of all fusion data used as the training set

[H] title	type	description	Measured
UKAEA_616.DEMO-HCPB-VV	UKAEA_FUS	DEMO-HCPB-VV spectra in 616 energy groups	M
UKAEA_150.NIF-ignition	UKAEA_FUS	NIF-ignition spectra in 150 energy groups	M
UKAEA_616.DEMO-HCPB-BP	UKAEA_FUS	DEMO-HCPB-BP spectra in 616 energy groups	M
UKAEA_616.WCLL-VV	UKAEA_FUS	WCLL-VV spectra in 616 energy groups	M
UKAEA_616.HCPB-VV	UKAEA_FUS	HCPB-VV spectra in 616 energy groups	M
UKAEA_616.WCCB-VV	UKAEA_FUS	WCCB-VV spectra in 616 energy groups	M
UKAEA_175.JAEA-FNS	UKAEA_FUS	JAEA-FNS spectra in 175 energy groups	M
UKAEA_175.ITER-DD	UKAEA_FUS	ITER-DD spectra in 175 energy groups	M
UKAEA_161.LMJ-g	UKAEA_FUS	LMJ-g spectra in 161 energy groups	M
UKAEA_175.ITER-DT	UKAEA_FUS	ITER-DT spectra in 175 energy groups	M
UKAEA_616.HCPB-FW	UKAEA_FUS	HCPB-FW spectra in 616 energy groups	M
UKAEA_616.DEMO-HCPB-FW	UKAEA_FUS	DEMO-HCPB-FW spectra in 616 energy groups	M
UKAEA_175.TUD-NG	UKAEA_FUS	TUD-NG spectra in 175 energy groups	M
UKAEA_616.WCLL-FW	UKAEA_FUS	WCLL-FW spectra in 616 energy groups	M

UKAEA_616_HCLL-FW	UKAEA_FUS	HCLL-FW spectra in 616 energy groups	M
-------------------	-----------	--------------------------------------	---

List of all fusion data used as the testing set

[H] title	type	description	Measured/Calculated
UKAEA_175_Frascati-NG	UKAEA_FUS	Frascati-NG spectra in 175 energy groups	M
UKAEA_100_JET-FW	UKAEA_FUS	JET-FW spectra in 100 energy groups	M
UKAEA_616_WCCB-FW	UKAEA_FUS	WCCB-FW spectra in 616 energy groups	M
UKAEA_616_HCLL-VV	UKAEA_FUS	HCLL-VV spectra in 616 energy groups	M