International Atomic Energy Agency

Nuclear Data Services

DOCUMENTATION SERIES OF THE IAEA NUCLEAR DATA SECTION

IAEA-NDS-0245

February 2024

Exploring TensorFlow for Nuclear Data Evaluation

Risa Kunitomo
Tokyo Institute of Technology, Tokyo, Japan

and

Georg Schnabel
IAEA, Nuclear Data Section, Vienna, Austria

Nuclear Data Section
International Atomic Energy Agency
Vienna International Centre, P.O. Box 100
A-1400 Vienna, Austria

E-mail: nds.contact-point@iaea.org
Fax: (43-1) 26007
Telephone: (43-1) 2600 21725
Web: https://nds.iaea.org

Disclaimer

Neither the author nor anybody else makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information disclosed, or represents that its use would not infringe privately owned rights.

Contents

1. Introduction

The objective of nuclear data evaluation is to produce reliable estimates and associated uncertainty information for fundamental nuclear quantities, such as cross sections and angular distributions. These data, compiled in nuclear databases, are essential for various fields within nuclear science and technology, such as nuclear energy, nuclear engineering, and nuclear medicine.

The nuclear data evaluation process comprises the collection of suitable experimental data, an assessment of their uncertainties, and the fusion of their information using a statistical method in order to obtain reliable estimates and associated uncertainties. Depending on the type of quantity and goal of the evaluation, sometimes predictions of nuclear physics models are also incorporated in this process.

Several evaluation codes exist that implement the statistical estimation procedure, such as GANDR [1] and GMAP [2]. These programs employ specific assumptions, such as basing the entire procedure on the multivariate normal distribution for modelling uncertainties. It is very difficult to generalize these codes to work with other probability distributions due to their complex code base and the need to implement support for these distributions from scratch. General frameworks for statistical modelling, such as STAN [3], require the formulation of the probabilistic assumptions in a programming language specific to these frameworks, posing an obstacle to converting the available nuclear data evaluation programs into this language. For these reasons, it is difficult to perform nuclear data evaluation with probability distributions different from the multivariate normal distribution and to include more fine-grained assumptions accurately reflecting the knowledge of evaluators.

TensorFlow[4] is a framework for machine learning usually employed for the creation of artificial neural networks and the training of their weights. A couple of years ago, Google released an extension to TensorFlow called TensorFlow-Probability [5]. This extension framework comes with support for various probability distributions whose parameters can be inferred by leveraging the powerful fitting algorithms provided by TensorFlow. Furthermore, TensorFlow does not come with its own programming language but rather provides functions that can be integrated into Python scripts. Therefore, TensorFlow-Probability may be a promising and flexible framework for performing nuclear data evaluations with probabilistic assumptions, that precisely reflect assumptions taken by an evaluator.

2. Purpose of this internship

The objective of this internship is to develop a basic understanding of the capabilities provided by TensorFlow and TensorFlow-Probability and to employ these packages in simple toy evaluation scenarios to learn more about their suitability for nuclear data evaluation.

3. Mathematical background and TensorFlow basics

Nuclear data evaluation is usually based on the assumption that the knowledge about nuclear quantities, such as cross sections, can be modelled by a multivariate normal distribution. This assumption was also taken for all the studies of this internship because the interface of Tensorflow-Probability seems to make it easy to switch to other distributions. The focus was therefore to recapitulate the common assumptions for nuclear data evaluation and on that basis develop Python scripts translating these assumptions into Tensorflow/Probability. The next section is concerned with a one-dimensional normal distribution to establish basic concepts. Afterwards, the discussion is extended to the multivariate normal distribution.

3.1. Normal distribution

A normal distribution is a probability distribution that is symmetric and bell-shaped. It is characterized by its probability density function (PDF) which describes the likelihood of observing a particular value. The two parameters characterizing a normal distribution are the mean value "μ" (representing the center of the bell-shaped form) and the standard deviation "σ" (defining its width). The normal PDF has the following functional form:

$$N(x,\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

Given that we know the mean value and the standard deviation, we can compute the likelihood of any possible observation "x".

3.1.1. A single normal distribution

At its core, nuclear data evaluation deals with the determination of best estimates and associated uncertainty information. For a one-dimensional normal distribution, these quantities are the mean value and the standard deviation.

As a starting point, we assume that we have a single observation represented by the value of $x = 3$ and that we also know the uncertainty (=standard deviation) $d = 1$ but we don't know the mean value μ. Therefore, we have $N(x,\mu,d) = N(3,\mu,1)$, which is now a function of μ. This very simple scenario captures the essence of the estimation problem encountered in nuclear data evaluation.

One wide-spread approach for finding unknown parameter values of a distribution is the Maximum Likelihood (ML) principle, which prescribes that the value of the unknown parameter should be selected so that the observed data becomes as likely as possible according to the PDF. Differently stated, we need to select the values of the unknown parameters that maximize the value of the PDF.

In the current simple scenario, we can solve this maximization problem analytically. Following, we describe the mathematical procedure to find the optimal value of μ so that we can compare it afterwards with the numerical solution obtained with the help of TensorFlow Probability.

First, we insert the specific values of x and d value into the normal PDF:

$$f(\mu) = N(3, \mu, 1) \ = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(3-\mu)^2}$$

Instead of working directly with this PDF, we can use its logarithm, which will not change the location of the maximum but simplify the mathematical operations to find it. Working with logarithmized PDFs also has a numerical advantage in computer programs because it protects against numerical underflow (exceeding the representational capabilities of floating point data types). After the application of the logarithm, we obtain the following functional form:

$$\log(N(3, \mu, 1)\ ) = -\frac{1}{2}\{(3-\mu)^2\} - \frac{1}{2}\log(2\pi)$$
$$= -\frac{1}{2}(\mu^2 - 6\mu + 9) - \frac{1}{2}\log(2\pi)$$

To find the maximum of the logarithmized PDF with respect to μ, we need to compute the first derivate and the value of μ where the first derivative becomes zero:

$$\frac{d}{d\mu}\log(N(3, \mu, 1)\ ) \ = -\frac{1}{2}(2\mu - 6) = 0$$
$$\therefore \mu = 3$$

According to the ML principle, $\mu = 3$ is the best value, which is not surprising given that the assumed observed value is $x = 3$.

We can use this analytical solution as a benchmark to verify that the numerical solution with TensorFlow yields the same result. In contrast to the analytical approach, we need to define a starting value for the unknown variable μ. The optimization procedure implemented with TensorFlow begins at this starting value and then uses the first derivative to make informed steps toward the maximum of the PDF. The iterative procedure is stopped once the change of μ from one iteration to the next falls below a numerical threshold or the derivative is sufficiently close to zero. The iterative optimization process is visualized in Fig. I.

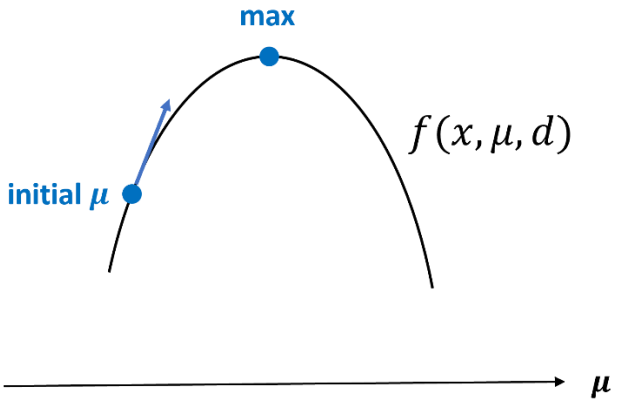

Figure I: Optimization of μ

We employed the “tfp.optimizer.bfgs_minimize” function for the numerical optimization.
Because we want to find a maximum, but this TensorFlow function can only identify a minimum we need to apply a mathematical workaround: We negated both the

normal PDF and its derivative function so that the maximization problem is converted to a minimization problem. This mathematical transformation is visualized in the following two figures.

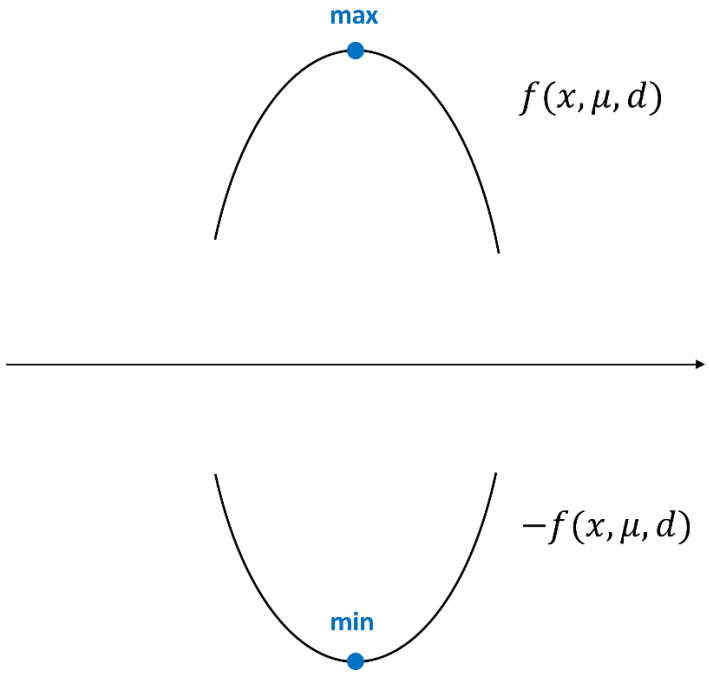

Figure II: Conversion of a maximization problem to a minimization problem by using the negated function.

The complete TensorFlow program implementing the optimization procedure is shown below. We set $\mu = 4$ as a start value. One powerful feature of TensorFlow is automatic differentiation. In contrast to the analytic approach explained above, we don't need to find the functional form of the derivative ourselves. This can be automatically accomplished by TensorFlow based on the PDF provided. Whereas it isn't a problem to find the form of the derivative analytically in our current simple scenario, it can become very difficult in more complicated cases with more complex distributions, such as the multivariate normal distribution discussed later in this report.
The part of the TensorFlow program that is concerned with the computation of the derivative is enclosed in the "tf.GradientTape" block. This tf.GradientTape construct is a context manager that plays a crucial role in automatic differentiation as it keeps track of intermediate results during the so-called backward pass through the computational graph. As we can see here, automatic differentiation is not only pertinent to find the weights of artificial neural networks with billions of weights but also in statistical inference with only a few unknown parameters.

```
import tensorflow as tf
import tensorflow_probability as tfp
import tensorflow.math as tfm
import math as m

def tfconst(value):
   return tf.constant(value, dtype=tf.float64)

def tfvar(value):
   return tf.Variable(value, dtype=tf.float64)
```

```
def f(x, mu, d):
   pi = tfconst(m.pi)
   y = 1. / tfm.square(2. * pi* d**2.) * tfm.exp(-0.5 * ((x - mu)/d)**2.)
   yy =tfm.log(y)
   return tf.squeeze(yy)

x = tfconst(3.)
d = tfconst(1.)
mu = tfvar(0.)

with tf.GradientTape() as tape:
   tape.watch(mu)
   y = f(x,mu,d)
g_mu = tape.gradient(y, mu)

def gradient(x, mu, d):
   with tf.GradientTape() as tape:
      tape.watch(mu)
      z= f(x,mu,d)
   g = tape.gradient(z, mu)
   return g

def value_and_gradients_function(mu):
   return (-f(x, mu, d), -gradient(x, mu, d))

start_mu = tfconst([4.])

optres = tfp.optimizer.bfgs_minimize(
   value_and_gradients_function,
   start_mu
)
print(optres)
```

The output of the TensorFlow program is provided below. The "position" variable indicates the solution found for the value of $\mu = 3$, consistent with the solution of the analytical approach. As can be seen, the output also contains detailed information about several other aspects of the optimization process. Notably, the variable "converged" (here True) indicates whether the function managed to locate the minimum. The variable "objective_value" contains the value of the function corresponding to the value for μ, which is in our case the negated value of the logarithmized normal PDF. Furthermore, "objective_gradient" contains the value of the derivative, which is zero, as expected at the maximum of the function.

```
BfgsOptimizerResults(converged=<tf.Tensor: shape=(), dtype=bool, numpy=True>,
failed=<tf.Tensor: shape=(), dtype=bool, numpy=False>, num_iterations=<tf.Tensor:
shape=(), dtype=int32, numpy=1>, num_objective_evaluations=<tf.Tensor: shape=(),
dtype=int32, numpy=3>, position=<tf.Tensor: shape=(1,), dtype=float64,
numpy=array([3.])>, objective_value=<tf.Tensor: shape=(), dtype=float64,
numpy=3.675754132818691>, objective_gradient=<tf.Tensor: shape=(1,), dtype=float64,
```

```
numpy=array([-0.])>, inverse_hessian_estimate=<tf.Tensor: shape=(1, 1), dtype=float64,
numpy=array([[1.]])>)
```

3.1.2. Estimation of μ based on two observations

We want to generalize the toy scenario to the case of two observations, to get one step closer to the case of a full nuclear data evaluation with potentially thousands of experimental data points. Let's assume that the two observations are given by the values 5 and 10. They are both described by one-dimensional normal distributions, $N(5, \mu, 1)$ and $N(10, \mu, 1)$. Assuming these two observations to be independent, the combined PDF is given by the product of the one-dimensional PDFs:

$$f(\mu) = N(5, \mu, 1) \times N(10, \mu, 1)$$
$$= \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(5-\mu)^2} \times \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(10-\mu)^2}$$
$$= \frac{1}{2\pi} e^{-\frac{1}{2}\{(5-\mu)^2+(10-\mu)^2\}}$$

We want to find the value of μ analytically for comparison with the numerical optimization using TensorFlow. Analogous to the case of a single observation, we first take the logarithm to simplify the problem:

$$\log\big(N(5, \mu, 1) \times N(10, \mu, 1)\big) = -\frac{1}{2}\{(5-\mu)^2 + (10-\mu)^2\} - \log(2\pi)$$
$$= -\frac{1}{2}(2\mu^2 - 30\mu + 125) - \log(2\pi)$$

Computing the derivative, setting it to zero and solving the resulting equation, we obtain

$$\frac{d}{d\mu}\log\big(N(5, \mu, 1) \times N(10, \mu, 1)\big) = -\frac{1}{2}(4\mu - 30) = 0$$
$$\therefore \mu = 7.5$$

Please note that this result is equal the arithmetic average, $5 + 10 / 2$. Therefore, we demonstrated that the Maximum Likelihood solution for two observations with the same standard deviation is given by a simple arithmetic average. It is not proven in this report, but this statement remains true for an arbitrarily large number of observations.

We adjusted the Python script for the case of one observation included in the previous section to the case of two observations. The modifications are performed in the "value_and_gradients" as well as in the "gradient" function by including the multiplication of two one-dimensional normal PDFs. As for the case of a single observation, we used the tfp.optimizer.bfgs_minimize function with the same mathematical trick to do the maximizaton. We adopted 60 as starting values for μ.

```
import tensorflow as tf
import tensorflow_probability as tfp
import tensorflow.math as tfm
import math as m
```

```
def f(x, mu, d):
   pi = tf.constant(m.pi,dtype=tf.float64)
   y = - tfm.log(tfm.square(2.*pi)) - tfm.log(d) - 0.5*((x - mu)/d)**2.
   return tf.squeeze(y)

def gradient(x, mu, d):
   with tf.GradientTape() as tape:
      tape.watch(mu)
      z = - f(x[0],mu,d[0]) * f(x[1],mu,d[1])
   g = tape.gradient(z, mu)
   return g

def value_and_gradients_function(mu):
   x = tf.constant([5.,10.], dtype=tf.float64)
   d = tf.constant([1.,1.], dtype=tf.float64)
   z = - f(x[0],mu,d[0]) * f(x[1],mu,d[1])
   return -z, -gradient(x, mu, d)

start_mu = tf.constant([60.], dtype=tf.float64)

optres = tfp.optimizer.bfgs_minimize(
   value_and_gradients_function,
   start_mu
)
print(optres)
```

The output of this program is included below. The result stored in the "position" variable reproduces the analytical solution.

```
BfgsOptimizerResults(converged=<tf.Tensor: shape=(), dtype=bool, numpy=True>, failed=<tf.Tensor: shape=(), dtype=bool, numpy=False>, num_iterations=<tf.Tensor: shape=(), dtype=int32, numpy=6>, num_objective_evaluations=<tf.Tensor: shape=(), dtype=int32, numpy=22>, position=<tf.Tensor: shape=(1,), dtype=float64, numpy=array([7.5])>, objective_value=<tf.Tensor: shape=(), dtype=float64, numpy=46.25025677505052>, objective_gradient=<tf.Tensor: shape=(1,), dtype=float64, numpy=array([-0.])>, inverse_hessian_estimate=<tf.Tensor: shape=(1, 1), dtype=float64, numpy=array([[0.90750628]])>)
```

3.1.3. Leveraging TensorFlow Probability's support for probability distributions

The previous sections demonstrated how probability distributions can be implemented from scratch using basic arithmetic operations and elementary functions, such as the exponential function. However, many commonly probability distributions are already implemented in TensorFlow-Probability and we can make use of them.

In particular, the one-dimensional normal distributions is available as tfp.distributions.Normal and can be instantiated with the following parameters:

```
tfp.distributions.Normal(
    loc,
    scale,
    validate_args=False,
    allow_nan_stats=True,
```

```
    name='Normal'
)
```

The parameter “loc” defines the mean value and the parameter “scale” the standard deviation of the normal distribution. The other parameters have default values and we can keep them.

We consider again the case of two observations with values 5 and 10, as in the previous section, and modify the Python script to make use of the tfd.Normal class provided by TensorFlow Probability:

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

def f(x, mu, d):
   dist = tfd.Normal(x, d)
   y = dist.prob(mu)
   return y

def gradient(x, mu, d):
   with tf.GradientTape() as tape:
      tape.watch(mu)
      z = - f(x[0],mu,d[0]) * f(x[1],mu,d[1])
   g = tape.gradient(z, mu)
   return g

def value_and_gradients_function(mu):
   x = tf.constant([5.,10.], dtype=tf.float64)
   d = tf.constant([1.,1.], dtype=tf.float64)
   z = - f(x[0],mu,d[0]) * f(x[1],mu,d[1])
   return -z, -gradient(x, mu, d)

start_mu = tf.constant([60.], dtype=tf.float64)

optres = tfp.optimizer.bfgs_minimize(
   value_and_gradients_function,
   start_mu
)
print(optres)
```

We obtain the same output as in 3.1.2, serving as additional validation that we’ve implemented the one-dimensional normal distribution correctly. Moreover, it also confirms that we are able to use the support for various distributions of TensorFlow Probability appropriately and we can proceed to a more complicated scenario, which involves the multivariate normal distribution.

3.2. Multivariate normal distribution

The case of two observations, where we used a product of two one-dimensional normal distributions can be regarded as a special case of the multivariate normal (MV) distribution. The MV distribution does not only allow us to conveniently model the PDF associated with an several (maybe even thousands of) data points but it also

enables the incorporation of correlations between distinct data points. The functional form of the MV PDF is given by:

$$f_X(x_1, x_2, \dots, x_k) = \frac{\exp\left(-\frac{1}{2}(x-\mu)^{\mathrm{T}}\Sigma^{-1}(x-\mu)\right)}{\sqrt{(2\pi)^k|\Sigma|}}$$

This distribution is characterized by a center vector μ and a covariance matrix Σ. If we know these two parameters, we can calculate the likelihood for any possible set of observations given in the vector x.

Having established confidence in the ability of TensorFlow to determine the correct mean value in the case of one and two observations, we now want to consider a slightly different estimation problem: We assume that we know the value of μ but don't know the uncertainty of the two experimental data points, which we want to estimate. We further assume that the unknown standard deviations of the two data points are the same. As an aside, in nuclear data evaluation, an analogous (but more complex) problem occurs when we have unknown experimental uncertainties and want to adjust them so that all experimental datasets are mutually consistent.

We are going to use the tfd.MultivariateNormalDiag function of TensorFlow Probability, which can be initialized using the following parameters:

```
tfp.distributions.MultivariateNormalDiag(
    loc=None,
    scale_diag=None,
    validate_args=False,
    allow_nan_stats=True,
    experimental_use_kahan_sum=False,
    name='MultivariateNormalDiag'
)
```

The "loc" parameter contains establishes the center vector and the "scale_diag" vector contains the diagonal elements of the covariance matrix Σ. Note that the diagonal elements are given by squared uncertainties (also called variances). The tfd.MultivariateNormalDiag class implements the case where all off-diagonal elements of the covariance matrix are zero. With this assumption, the PDF of the multivariate normal PDF is given by a product of the one-dimensional normal distributions, each associated with an observed value in x. Using this simplifying assumption, it will be easier to assess the correctness of the result produced by TensorFlow.

We assume that the values of the two observations are 1 and 5, hence $x = (1, 5)$, and that the center vector μ is given by $(3, 3)$. Further we assume that the diagonal elements dar are given by (d, d), hence both variances having the same value. We set $d = 3$ as starting value. The following TensorFlow program implements the ML principle by maximizing the multivariate normal PDF with respect to d.

```
import tensorflow as tf
import tensorflow_probability as tfp
import numpy as np
```

```
import math
tfd = tfp.distributions

def f(x, mu, dar):
   dar = tf.repeat(dar, tf.size(mu))
   mvn = tfd.MultivariateNormalDiag(mu, dar)
   y = mvn.log_prob(x)
   return y

def gradient(x, mu, dar):
   with tf.GradientTape() as tape:
      tape.watch(dar)
      ff = f(x, mu, dar)
   g = tape.gradient(ff, dar)
   return g

@tf.function
def value_and_gradients_function(d):
   x = tf.constant([1.,5.], dtype=tf.float64)
   mu = tf.constant(3., dtype=tf.float64)
   mu = tf.repeat([mu], tf.size(x))
   dar = tf.math.abs(d)
   print(dar)
   res, grad = -f(x, mu, dar), -gradient(x, mu, dar)
   return res, grad

di=3
start = tf.constant([di], dtype=tf.float64)

optres = tfp.optimizer.bfgs_minimize(
   value_and_gradients_function,
   initial_position=start,
   )
print(optres)
```

The output is shown below, indicating the result $d = 2$ (in the "position" variable).

```
BfgsOptimizerResults(converged=<tf.Tensor: shape=(), dtype=bool, numpy=True>, failed=<tf.Tensor: shape=(), dtype=bool, numpy=False>, num_iterations=<tf.Tensor: shape=(), dtype=int32, numpy=6>, num_objective_evaluations=<tf.Tensor: shape=(), dtype=int32, numpy=16>, position=<tf.Tensor: shape=(1,), dtype=float64, numpy=array([2.])>, objective_value=<tf.Tensor: shape=(), dtype=float64, numpy=4.224171427529236>, objective_gradient=<tf.Tensor: shape=(1,), dtype=float64, numpy=array([8.8817842e-16])>, inverse_hessian_estimate=<tf.Tensor: shape=(1, 1), dtype=float64, numpy=array([[1.0025981]])>)
```

4. Nuclear data evaluation

The process of nuclear data evaluation involves gathering appropriate experimental data, evaluating their uncertainties, and integrating their information through statistical methods to derive dependable estimates and associated uncertainties. In the following sections, we explore the application of TensorFlow in slightly more realistic scenarios

by using real (meaning not synthetic) experimental data. We will make use of the basic building blocks provided by TensorFlow and TensorFlow Probability, introduced in the previous section, for this purpose.

4.1. Retrieval of experimental data from the EXFOR database

We created a Python script for retrieving cross section measurements from the EXFOR database [6]. The Experimental Nuclear Reaction Database (EXFOR) is a comprehensive collection of experimental datasets with measurements of nuclear quantities, such as cross sections, angular distributions, and energy spectra.
In our Python script, we relied on EXFORTABLES [7] created by Shin Okumura, which provides cross sections in tabulated form and is derived from the original EXFOR database by using a parsing package written in Python. One objective of this package is to facilitate the access to the experimental data in order to enable the application of advanced analysis methods, such as those employed in machine learning for the discovery of hidden patterns.

The tabular format provided by EXFORTABLES is of the simple form (x, y, dx, dy) with x being the incident energy, y the cross section and dx and dy the respective uncertainties. These files with tabular data in a nested directory structure with different levels of the hierarchy indicating the incident particle, nuclide and reaction. An example file of EXFORTABLES is shown below:

```
# entry-subent-pointer  : 11180-006-0
# EXFOR reaction       : ['26-FE-56', ['N,TOT'], ',,SIG']
# incident energy      : 1.0000e-05 MeV - 2.8000e-03 MeV
# target             : Fe-56
# product            : -
# level energy       : -
# MF-MT number       : 3 - 1
# first author       : C.T.Hibdon
# institute          : (1USAANL): Argonne National Laboratory, Argonne, IL
# reference          : (P,ANL-4963,3,195301)
# year               : 1953
# facility           : (VDG): Van de Graaff
# git                : https://github.com/IAEA-
NDS/exfor_master/blob/main/exforall/111/11180.x4
# nds                : https://nds.iaea.org/EXFOR/11180
#
#     E_in(MeV)      dE_in(MeV)         XS(B)         dXS(B)
    1.00000E-05    0.00000E+00    1.31000E+01    1.00000E+00
    2.00000E-05    0.00000E+00    1.20000E+01    1.00000E+00
    1.26000E-04    0.00000E+00    1.20000E+01    2.00000E+00
    3.45000E-04    0.00000E+00    1.00000E+01    2.00000E+00
    2.80000E-03    0.00000E+00    1.00000E+01    3.00000E+00
```

The following code snippet shows the implementation of the function to extract the cross sections (given in column XS) and the associated energies (given in column E_in) from EXFORTABLES. We employed “pandas” for combining all EXFORTABLES text files, and “os” for locating all files with relevant nuclear data.

```
import tensorflow as tf
import numpy as np
import os
import pandas as pd

def get_experimental_data(path, file_list):

   df2 = pd.DataFrame()

   for e in file_list:
      info = e.split("_")
      target = info[0]
      reaction = info[1]
      bib = info[2]
      author, entry, subent, pointer, year = bib.split("-")
      filename = os.path.join(path, e)
      #print(filename)

      df = pd.read_csv(filename,
                  sep="\s+",
                  index_col=None,
                  header=None,
                  usecols=[0, 1, 2, 3],
                  comment="#",
                  names=["Energy", "dE", "XS", "dXS"],
                  )
      df["author"] = author
      df["year"]= year

      df2 = pd.concat([df, df2])

   file_list = np.array(df2["author"])
   E_dash = np.array(df2["Energy"], dtype=np.float64)
   XS = np.array(df2["XS"], dtype=np.float64)
   return file_list, E_dash, XS
```

4.2. Linear interpolation

Given a computational mesh of energies and associated cross sections, linear interpolation can be used to compute the cross-section values of energies that lie in-between the mesh points. The underlying assumption is that the functional form between two consecutive mesh points can be described as a straight line.
To develop a better intuition, consider the following visual interpretation of the interpolation procedure shown in Fig. III. Imagine you have two points on a line, and you want to find a point in between them. You would pick the point on the line that corresponds to a certain x-value (indicated by E' in the figure) and then determine the corresponding y-value by following the imagined line parallel to the x-axis until you arrive at the y-axis showing the value (indicated by μ' in the figure).

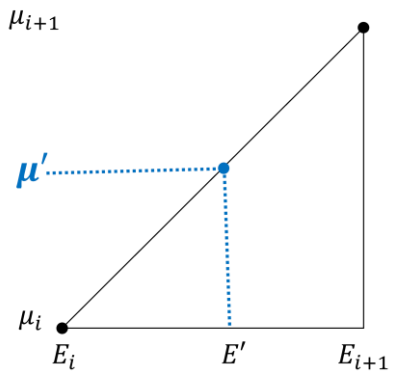

Figure III: Linear interpolation

Linear interpolation is useful in nuclear data analysis and visualization to fill in missing data points between known values. This is especially helpful in creating smooth curves and trends when plotting graphs. If we know $E_i, E_{i+1}, \mu_i, \mu_{i+1}$, and E', we can find μ' according to E'. The formula to perform linear interpolation is given by:

$$\mu' = \mu_i + \frac{\mu_{i+1} - \mu_i}{E_{i+1} - E_i}(E' - E_i)$$

The values E_i and E_{i+1} are consecutive energies on the computational mesh and μ_i and μ_{i+1} the corresponding cross section values. The cross-section value μ' is associated with the energy E' lying in-between E_i and E_{i+1}.

For the implementation of linear interpolation with TensorFlow, we relied on the ”tf.searchsorted” function. It is used to perform binary search in a sorted array or tensor. “tf.gather” is a function in TensorFlow that allows you to gather slices from a tensor along a specified axis. In the case of a vector (a one-dimensional tensor), it helps to retrieve specific elements based on the provided indices. The Python function making use of the TensorFlow functions to implement linear interpolation is given by:

```
def linearinterpol(E, mu, E_dash):

    ivec = tf.searchsorted(E, E_dash) - 1
    mu_i = tf.gather(mu, ivec)
    mu_ip1 = tf.gather(mu, ivec+1)
    E_i = tf.gather(E, ivec)
    E_ip1 = tf.gather(E, ivec+1)

    coeff = (mu_ip1-mu_i)/(E_ip1-E_i)
    mu_dash = mu_i + coeff * (E_dash-E_i)
    return mu_dash
```

4.3. Linear interpolation of nuclear data to find the best μ

Having explained linear interpolation in general, we want to elaborate on how it can be used in the context of nuclear data evaluation. For nuclear data evaluation, one typically introduces a computational mesh with the theoretical cross section values. However,

experimental data are given at different energies and hence we need to propagate the theoretical cross section values to the experimental energies. We can achieve this propagation by using linear interpolation as implemented in our Python script above. More precisely, we have a theoretical mesh, denoted as E, and a vector μ containing cross-sections at energies listed in E. Subsequently, we set up the vector E' with experimental energies and the vector XS with the corresponding cross-section values.
Once we have obtained the propagated cross section values μ', we can evaluate the multivariate normal distribution function, $N(XS, \mu', dar)$. The chain of operations is visualized as a graph in Fig. IV.

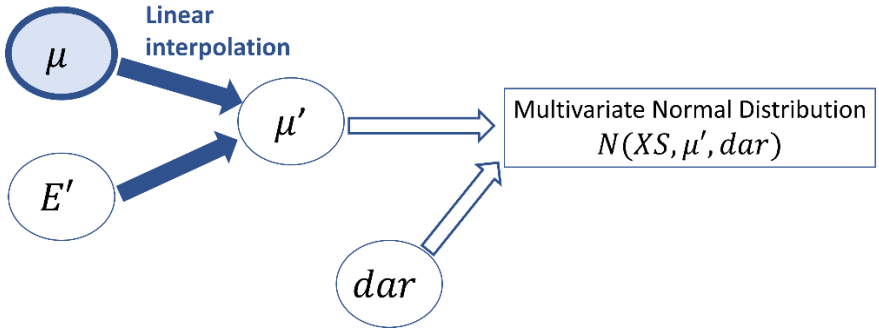

Figure IV: Linear interpolation to find the best μ

As in an earlier toy scenario, we employ the function tfd.MultivariateNormalDiag. TensorFlow enables us to find appropriate values in the vector μ and/or in the vector d by optimization.

5. Demonstration of approach

Now we combine the retrieval of experimental data, linear interpolation and the multivariate normal distribution to perform a schematic nuclear data evaluation. In different scenarios, we explore the determination of the mean vector μ, and the diagonal elements in the covariance matrix dar.

5.1. Optimization of a scale vector μ of the multivariate normal distribution

We want to employ a similar approach as discussed in section 4.3. to find the best μ using linear interpolation. We assume that we know E', XS values because they can be retrieved from the EXFORTABLES database. We further assume a computational mesh of incident energies E with 15 mesh points equally spaced between 0 and 60 MeV, and that every experimental data point is associated with the same uncertainty (standard deviation) given by $d = 1$. Our goal is to find the optimal value for μ.
For the demonstration we consider the neutron-induced total cross section of Fe-56. Fe-56 is an important structural material used in the construction of fission and fusion reactors and experiments. Especially, the total cross section is important for calculation of material damage. This damage is caused by the interaction of high-energy neutrons that may be produced by D-T reactions within the plasma in a fusion device.
In the following script, we can define the "path" to point to the folder of EXFORTABLES containing the total cross section of Fe-56. The script loads the experimental data and then determines the optimal values μ on the computational mesh. To achieve this, it employs the "tfp.optimizer.bfgs_minimize" function, linear interpolation to propagate the theoretical values the experimental values and the tfd.MultivariateNormalDiag TensorFlow function to compute value of the PDF for a

given choice of the vector μ. The bfgs_minimize function implements an iterative procedure to refine the values in μ in order to maximize the corresponding value of the PDF. We use as starting point "start_mu" a vector that is filled with the value 60.

```
import tensorflow as tf
import tensorflow_probability as tfp
import os
import numpy as np
tfd = tfp.distributions
import matplotlib.pyplot as plt
from expdata_utils import linearinterpol
from expdata_utils import get_experimental_data

path = "D:/nucleardata/exfortables_py/n/Fe-56/n-tot/xs/"
exfiles = os.listdir(path)
file_list, E_dash, XS = get_experimental_data(path, exfiles)
E = np.linspace(0, 60, 15)

def f(x, mu, dar):
   mu_dash = linearinterpol(E, mu, E_dash)
   dar = tf.repeat(dar, tf.size(x))
   mvn = tfd.MultivariateNormalDiag(mu_dash, dar)
   y = mvn.log_prob(x)
   return y

def gradient(x, mu, dar):
   with tf.GradientTape() as tape:
      tape.watch(mu)
      ff = f(x, mu, dar)
   g = tape.gradient(ff, mu)
   return tf.convert_to_tensor(g)

@tf.function
def value_and_gradients_function(mu):
   dar = tf.constant([1.], dtype=tf.float64)
   res, grad = -f(XS, mu, dar), -gradient(XS, mu, dar)
   return res, grad

def find_optimized_mu(start_mu):
   optres = tfp.optimizer.bfgs_minimize(
      value_and_gradients_function,
      initial_position = start_mu,
      # tolerance=1e-2
   )
   print(optres)
   return optres.position

if __name__ == "__main__":
   start_mu = tf.constant([60.]*len(E), dtype=tf.float64)   # mu
   optimized_mu = find_optimized_mu(start_mu)
   plt.plot(E, start_mu)  # blue
   plt.scatter(E_dash, XS)  # blue dots
   plt.plot(E, optimized_mu)  # orange
```

```
#plt.xlim([5,55])
#plt.ylim([0,10])
plt.show()
```

The following graph shows the result of the optimization procedure. The blue line corresponds to the initial vector "start_mu" (filled with the value 60) containing the corresponding cross section value for each energy of the computational mesh. The blue dots indicate experimental data points. The orange line indicates the result given by optimized values in "mu", with intermediate values obtained by linear interpolation for displaying a continuous curve. We see that the optimized values revert back to the starting value above 55 MeV because there is no data to cause the optimization procedure to modify the starting values.

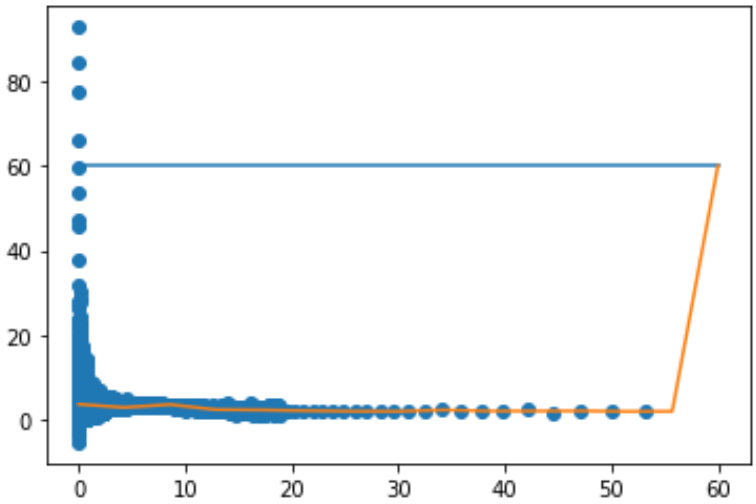

Figure V: Comparison of optimization result with experimental data

The following plot shows the same result and data but is restricted to the area $5 \leq x \leq 55$ and $0 \leq y \leq 10$ for a more detailed display. The obtained solution aligns well with the experimental data.

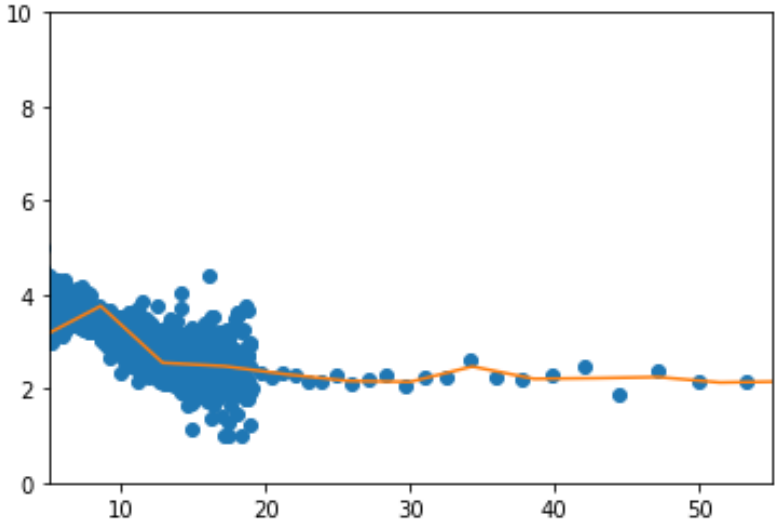

Figure VI: Comparison of optimization result with experimental data restricted to the plotting region $[5 \leq x \leq 55, 0 \leq y \leq 10]$

5.2. Obtaining uncertainties d of the multivariate normal distribution

Now we want to consider the case where we know neither the vector μ nor the variance d.

To address this case, we calculate first the optimized μ using the code of the previous section. Then we keep this obtained value fixed and optimize the function again, but now with respect to d. We further assume two different possible assumptions: 1) Same uncertainty d for all data points, and 2) an individual uncertainty d for each data point. The relationship between the variables is depicted in Fig. VII.

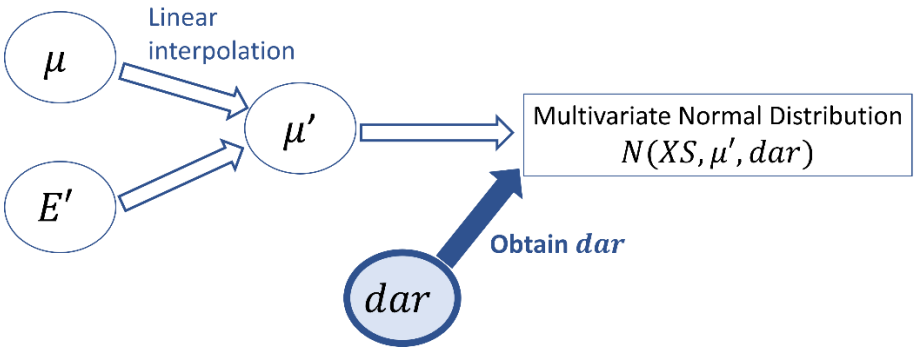

Figure VII: The relationship of variables involved in the optimization process

5.2.1. Same uncertainty d for all data points

Let's first discuss the case where all experimental uncertainties are the same. In the Python script, this assumption can be implemented by repeating a single value to fill the vector dar whose length is given by the number of experimental data points. We implemented this assumption by calling "tf.repeat" inside the function $f_d(x, \mu, d)$. Otherwise, the following Python script to implement this scenario is structured in a similar way as the examples already discussed before.

```
import tensorflow as tf
import tensorflow_probability as tfp
import os
import numpy as np
tfd = tfp.distributions
from expdata_utils import linearinterpol
from expdata_utils import get_experimental_data
from linearinterpol_Fe56 import find_optimized_mu
import matplotlib.pyplot as plt

path = "D:/nucleardata/exfortables_py/n/Fe-56/n-tot/xs/"
exfiles = os.listdir(path)
file_list, E_dash, XS = get_experimental_data(path, exfiles)

# select expdata in a specific energy interval
sel = (E_dash > 10) & (E_dash < 20)
E_dash = E_dash[sel]
XS = XS[sel]

E = np.linspace(0, 60, 10)

def f_d(x, mu, d):
```

```
    # use this for individual d for each data point
    # dar = d
    ## or this for the same d value for all data points
    dar = tf.repeat([d], tf.size(XS))
    mu_dash = linearinterpol(E, mu, E_dash)
    mvn = tfd.MultivariateNormalDiag(mu_dash, dar)
    y = mvn.log_prob(x)
    return y

def gradient_d(x, mu, d):
    with tf.GradientTape() as tape:
        tape.watch(d)
        ff = f_d(x, mu, d)
    g = tape.gradient(ff, d)
    return tf.convert_to_tensor(g)

@tf.function
def value_and_gradients_function_d(d):
    dabs = tf.abs(d)
    res, grad = -f_d(XS, optimized_mu, dabs), -gradient_d(XS, optimized_mu, dabs)
    return res, grad

def find_optimized_d(start_d):
    start_d = tf.reshape(start_d, (-1,))
    optres = tfp.optimizer.bfgs_minimize(
        value_and_gradients_function_d,
        initial_position = start_d,
        # tolerance=1e-2
    )
    print(optres)
    return optres.position

if __name__ == "__main__":
    start_mu = tf.constant([5.]*len(E), dtype=tf.float64)   # mu
    optimized_mu = find_optimized_mu(start_mu)
    start_d = tf.constant(1., dtype=tf.float64)   # d
    # res = value_and_gradients_function_d(start_d)
    # use the following only for individual d values for each data point
    # start_d = tf.repeat([start_d], tf.size(XS))
    optimized_d = find_optimized_d(start_d)

    plt.errorbar(E_dash, XS, yerr=optimized_d, fmt="o",
                 alpha=0.2)
    plt.plot(E, optimized_mu)
    plt.xlim([10, 20])
    plt.ylim([0, 10])
    plt.xlabel("energy [MeV]")
    plt.ylabel("cross section [barn]")
    plt.show()
```

The result is shown in Fig. VIII. Clearly, some error bars are too short to be consistent with the evaluated curve. Under the constraint that all error bars must be of the same size, TensorFlow finds a compromise between data points that are very close to the

evaluation and those very far away. This leads to the observed result that for the extreme outliers the “compromise” error bars are not large enough. Therefore, we also studied the scenario where each data point can have its individual uncertainty, demonstrated in the next section.

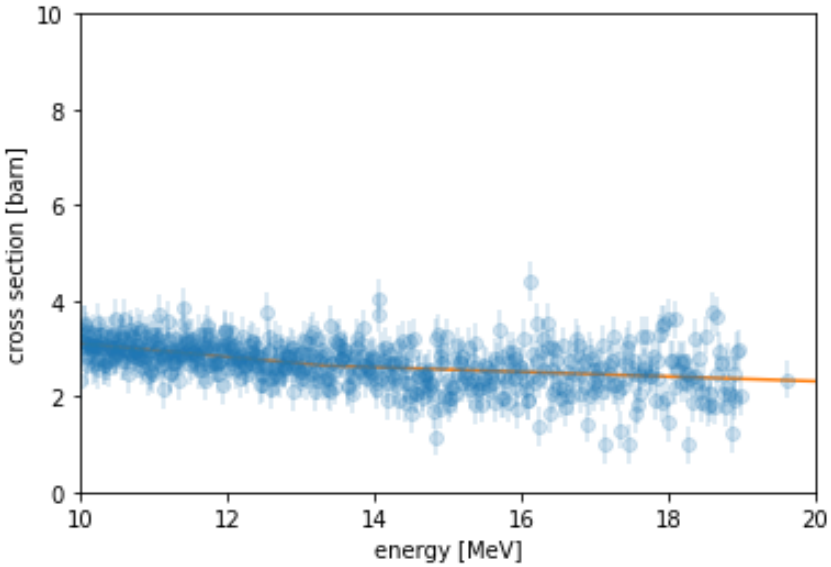

Figure VIII: Same uncertainty d for all data points

5.2.2. Individual uncertainties d for each data point

Now we allow for each data point having its own individual uncertainty d. The necessary modifications to implement this scenario can be seen in the script provided below. Especially, consider the changes in the main block of the program under the “if __name__ == "__main__":” statement.

```
import tensorflow as tf
import tensorflow_probability as tfp
import os
import numpy as np
tfd = tfp.distributions
from expdata_utils import linearinterpol
from expdata_utils import get_experimental_data
from linearinterpol_Fe56 import find_optimized_mu
import matplotlib.pyplot as plt

path = "D:/nucleardata/exfortables_py/n/Fe-56/n-tot/xs/"
exfiles = os.listdir(path)
file_list, E_dash, XS = get_experimental_data(path, exfiles)

# select expdata in a specific energy interval
sel = (E_dash > 10) & (E_dash < 20)
E_dash = E_dash[sel]
XS = XS[sel]

E = np.linspace(0, 60, 10)

def f_d(x, mu, d):
   # use this for individual d for each data point
```

```
    dar = d
    ## or this for the same d value for all data points
    ## dar = tf.repeat([d], tf.size(XS))
    mu_dash = linearinterpol(E, mu, E_dash)
    mvn = tfd.MultivariateNormalDiag(mu_dash, dar)
    y = mvn.log_prob(x)
    return y

def gradient_d(x, mu, d):
    with tf.GradientTape() as tape:
        tape.watch(d)
        ff = f_d(x, mu, d)
    g = tape.gradient(ff, d)
    return tf.convert_to_tensor(g)

@tf.function
def value_and_gradients_function_d(d):
    dabs = tf.abs(d)
    res, grad = -f_d(XS, optimized_mu, dabs), -gradient_d(XS, optimized_mu, dabs)
    return res, grad

def find_optimized_d(start_d):
    start_d = tf.reshape(start_d, (-1,))
    optres = tfp.optimizer.bfgs_minimize(
        value_and_gradients_function_d,
        initial_position = start_d,
        # tolerance=1e-2
    )
    print(optres)
    return optres.position

if __name__ == "__main__":
    start_mu = tf.constant([5.]*len(E), dtype=tf.float64)   # mu
    optimized_mu = find_optimized_mu(start_mu)
    start_d = tf.constant(1., dtype=tf.float64)   # d
    # res = value_and_gradients_function_d(start_d)
    # use the following only for individual d values for each data point
    start_d = tf.repeat([start_d], tf.size(XS))
    optimized_d = find_optimized_d(start_d)

    plt.errorbar(E_dash, XS, yerr=optimized_d, fmt="o",
                 alpha=0.2)
    plt.plot(E, optimized_mu)
    plt.xlim([10, 20])
    plt.ylim([0, 10])
    plt.xlabel("energy [MeV]")
    plt.ylabel("cross section [barn]")
    plt.show()
```

The following plot show the result for the case of individual uncertainties d for each data point. The error bars of data points further away from the evaluation are larger than those for the points closer to it. This is in contrast to the result shown in 5.2.1 where all error bars were of equal length.

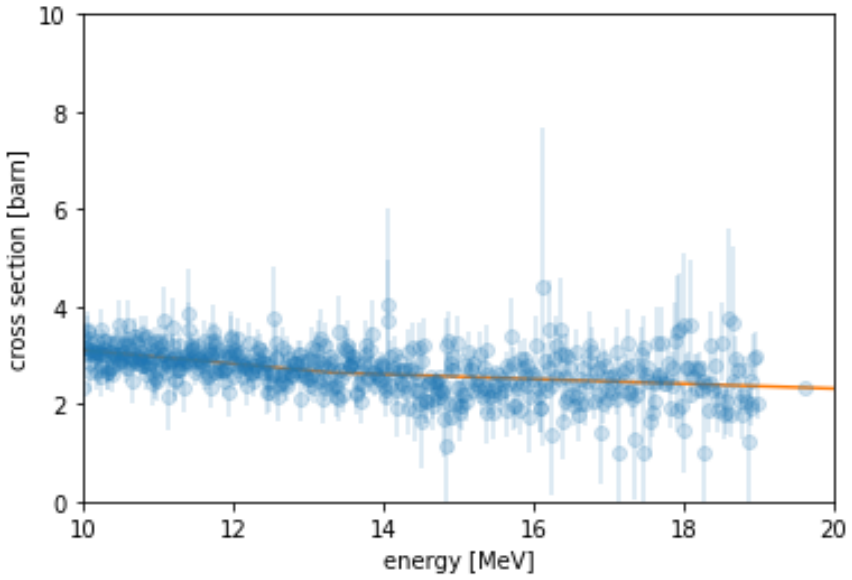

Figure IX: Individual uncertainties d for each data point

6. Summary

The objective of this internship was to explore the suitability of the TensorFlow framework in combination with the TensorFlow Probability extension package for the purpose of nuclear data evaluation. We based our investigation on the commonly used assumption that the experimental data and their uncertainties can be modelled by (multivariate) normal distributions. As the core task of nuclear data evaluation is to find best estimates and associated uncertainties, we started with the investigation of simple toy scenarios using the normal distribution and only one or two observations. In these cases, we were able to find analytic solutions to the estimation problems, which served as benchmarks for validating the solutions found by numerical optimization with TensorFlow. We created Python scripts leveraging TensorFlow probability to solve these estimation problems and obtained solutions that were consistent with the analytical solutions.

After we have implemented the probability density functions (PDF) of the normal distribution ourselves, we instead adopted the corresponding ready-made distribution function provided by TensorFlow Probability and verified their correct usage by comparing to the result of one of the simple toy cases before. In general, the ability to leverage ready-made implementations of distributions is essential because realistic nuclear data evaluation scenarios may require more complicated distributions and we want to avoid the need to implement them ourselves.

After the investigation of these simplistic toy scenarios, we gradually approached more realistic evaluation scenarios. We managed to create a Python script to retrieve neutron-induced total cross section data for Fe-56 from the EXFOR database. For this purpose, we leveraged the convenient access to the database provided by EXFORTABLES, which exposes the data in tabular files under an intuitive directory structure on the computer.

Usually, one uses a computational energy mesh with corresponding cross sections to be estimated and experimental data given on a different energy mesh. Therefore, we

discussed linear interpolation and how it can be used to propagate function values from a computational mesh to the mesh of the experimental data. We then implemented linear interpolation as a Python function using functionality from TensorFlow.

Finally, we combined all the developed building blocks, which are 1) the capability to estimate unknown parameters of a PDF, 2) linear interpolation to propagate cross section values from a computational energy mesh to the experimental energies, and 3) the ability to retrieve experimental data, to consider an evaluation scenario that is not so different anymore from a real nuclear data evaluation scenario.

Using experimental data for the neutron-induced total cross section of Fe-56, we estimated an evaluated curve (the center vector of a multivariate normal distribution) based on the experimental data points and assuming that all data points are affected by the same and known uncertainty. In a second step, we relied on the obtained evaluated curve to estimate the uncertainties of the experimental data points. We studied two cases: 1) All of the experimental data points are affected by the same unknown uncertainty, and 2) Each data points can have a different uncertainty. We were successfully able to determine the solution for both cases. We briefly discussed the features of the plots showing the results, especially how the different assumptions affect the uncertainties.

In summary, over the course of the internship, we were able to go from the mathematical foundations to an (almost realistic) evaluation scenario. We conclude from our investigation that TensorFlow in combination with TensorFlow Probability is indeed a powerful framework that can be leveraged to perform nuclear data evaluation. Even though, we considered simplified scenarios, it is conceptually straight-forward to extend the assumptions by making use of other distributions or optimization algorithms provided by TensorFlow and TensorFlow probability.

7. Bibliography

[1] GANDR: https://nds.iaea.org/gandr/

[2] GMAP: https://nds.iaea.org/standards/codes.html

[3]: STAN: https://mc-stan.org/

[4] TensorFlow: https://www.tensorflow.org/

[5] TensorFlow Probability: https://www.tensorflow.org/probability

[6] Otuka, N. et al. "Towards a More Complete and Accurate Experimental Nuclear Reaction Data Library (EXFOR): International Collaboration Between Nuclear Reaction Data Centres (NRDC)". Nuclear Data Sheets 120 (2014), pp. 272–276. issn: 00903752. doi: 10.1016/j.nds.2014.07.065.

[7] Okumura, S., Schnabel, G., and Koning, A. "Development of an EXFORTABLES-Inspired Structured Database through EXFOR Parser". Proceedings of the 2024 Symposium on Nuclear Data and PHITS, November 15-17, 2023, Tokai, Japan, (submitted).