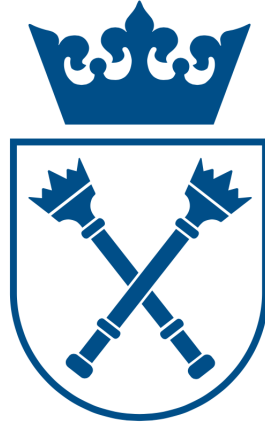


Jagiellonian University in Kraków  
Faculty of Physics, Astronomy and Applied Computer Science



**Diana Janik**

Index number: 1103340

**Neural networks application in hazardous materials  
recognition**

Master's thesis in Applied Computer Science

Master thesis advisor:  
Dr. Michał Silarski  
Department of Experimental  
Particle Physics and Applications

Kraków 2020

## Abstract

This work presents the feasibility studies of mustard gas recognition using neural network algorithms and data samples measured by a novel, non-invasive device, which is being developed at the Jagiellonian University within the framework of the SABAT (Stoichiometry Analysis By Activation Techniques) project. The data samples used to train and validate the performance of the used algorithms were based on realistic Monte Carlo simulations which formed histograms of energy depositions for three intervals of detection gamma quanta time. Multiple neural network models have been trained and tested using 7-folds cross-validation in order to analyse how does detector's sensitivity impact model's precision. The best results have been achieved for the  $\text{LaBr}_3\text{:Ce}$  detector. Nonetheless, based on simulated data from inexpensive, less sensitive  $\text{NaI:Tl}$  detector obtained model's accuracy has been only slightly lower. It is expected, that training a model on a larger dataset, without a burden of correlation caused by a limitation in the size of the simulations, may improve the results for  $\text{NaI:Tl}$  detector.

*I would like to express my sincere gratitude*

*to my supervisor, dr. Michal Silarski,  
for his patient guidance, support  
and encouragement throughout this project,*

*to Sushil Sharma  
for his invaluable help  
and providing the simulation data.*

# Contents

<b>1</b>	<b>Introduction</b>	<b>8</b>
<b>2</b>	<b>Neural Networks</b>	<b>11</b>
2.1	Biological neuron . . . . .	11
2.2	Mathematical model of neuron . . . . .	12
2.2.1	Activation functions . . . . .	12
2.3	Perceptron . . . . .	14
2.4	Feedforward Neural Network . . . . .	14
2.5	Neural network learning process . . . . .	14
2.5.1	Binary cross-entropy . . . . .	16
2.5.2	Hinge Loss . . . . .	17
2.5.3	Hinge squared Loss . . . . .	18
2.5.4	R1 regularization . . . . .	18
2.5.5	R2 regularization . . . . .	19
2.5.6	Optimization algorithms . . . . .	19
2.5.7	Glorot and He weights initialization . . . . .	23
2.5.8	Batch Normalization . . . . .	24
<b>3</b>	<b>Data preparation</b>	<b>25</b>
<b>4</b>	<b>Trained models and their performance</b>	<b>31</b>
<b>5</b>	<b>Conclusions</b>	<b>35</b>
<b>6</b>	<b>Summary</b>	<b>36</b>
<b>A</b>	<b>Python code</b>	<b>40</b>
A.1	Data preparation . . . . .	40
A.1.1	Gaussian smearing . . . . .	40
A.1.2	Histograms generation . . . . .	42
A.1.3	Utils . . . . .	45
A.2	Training models . . . . .	46
A.2.1	Dataset preparation . . . . .	47
A.2.2	Visualization . . . . .	51
A.2.3	Trainig and testing models . . . . .	52

## List of Figures

1	A schematic representation of principles of the NAA method. A neutron scatters inelastically with an atomic nucleus leading to its excitation. Transition to the ground state of the nucleus leads to gamma ray emission which is registered by a detector [2].	8
2	Scheme of the simulated SABAT system investigating mustard gas in aquatic environment [3]. . . . .	9
3	A schematic representation of a basic nerve cell. Adapted from [10]. . . . .	11
4	Scheme of a mathematical model of a neuron. It consists of input vector $x = (x_0, x_1, x_2, \dots, x_n)$ , weights $w = (w_0, w_1, w_2, \dots, w_n)$ , a net input function denoted as $\Sigma$ , and an activation function $f(v)$ [12]. . . . .	12
5	Schema of a perceptron consisting of 2 input and 3 output neurons. Adapted from [13].	14
6	Pictorial representation of the underfitted (left plot), overfitted (right plot) and properly trained neural networks. In the plots, the crosses and the circles represent data points of two different classes, and the dashed line illustrates a pattern of a model. Adapted from [16]. . . . .	15
7	Qualitative dependence of the neural network performance as a function of the complexity. The optimum performance is achieved by minimization of the total error defined as the sum of variance, the bias squared and so called irreducible error, which is a measure of noise in the data [18]. . . . .	16
8	A standard neural network without dropout (a) and the neural network with two omitted neurons at the training stage with applied dropout (b) [24]. . . . .	19
9	Example of a loss function with saddle point, local minimum, and global minimum [26].	20
10	Gradient descent with different learning rates: too small learning rate (left plot) require a lot of learning epochs before reaching a minimum, accurate learning rate (middle plot) enables to step into a minimum in optimal number of iterations, and too big learning rate (right plot) causes moving too far away in each of a step [26]. .	21
11	Comparison of Gradient Descent and Stochastic Gradient Descent convergence. The GD algorithm (left plot) in each iteration converges slightly better than the SGD algorithm (right plot). However, computation cost of each iteration of the SGD is much lower than the GD [26]. . . . .	21
12	Energy deposition and global time in mustard gas and background datasets. . . . .	25
13	Schema of inelastic scattering and neutron capture processes. . . . .	26
14	Simulated distributions of energies measured by the detector for three bins of gamma quanta registration time. . . . .	26
15	Full-width-at-half-maximum illustrated at a Gaussian-shaped peak [34]. . . . .	27
16	Energy resolution [%] and FWHM [MeV] for detectors. . . . .	28

17	Exemplary simulated data samples based on randomly chosen events and represented as histograms of energy deposition: a) without taking into account gamma quanta detector resolution, sample of $2 \cdot 10^4$ events, b) with energy resolution corresponding to the $\text{LaBr}_3\text{:Ce}$ detector ( $2 \cdot 10^4$ events), c) applying Gaussian smearing using parametrization for the $\text{NaI:Tl}$ detector ( $2 \cdot 10^4$ events), d) without taking into account gamma quanta detector resolution ( $5 \cdot 10^3$ events). . . . .	29
18	Example data samples based on $5 \cdot 10^3$ randomly chosen simulated events represented as histograms of the energy deposition for three slots of the gamma quanta registration time. Histograms were done without energy smearing ( a ) ) and applying energy resolution parametrization for b) $\text{LaBr}_3\text{:Ce}$ , c) $\text{NaI:Tl}$ , d) BGO and e) LSO materials.	30
19	Architecture of the neural network chosen to be used as classifier for the SABAT data analysis. . . . .	32

## List of Tables

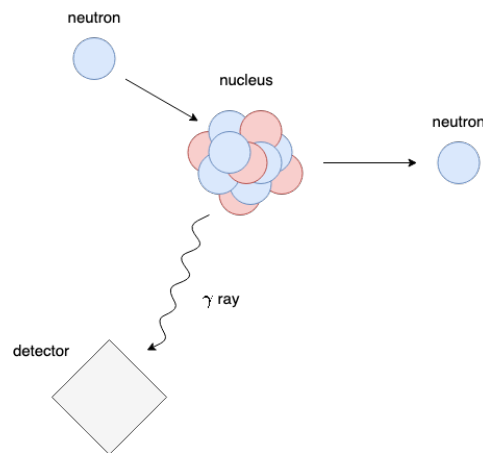
1	Selected activation functions used with supervised neural networks. . . . .	13
2	Glorot and He initializations parameters for normal and uniform distribution, where the number of incoming and outgoing connections of a layer, which weights are being initialized are denoted as $fan\_in$ and $fan\_out$ , respectively [13]. . . . .	23
3	Energy resolution coefficients for most commonly used scintillating detectors. The details of measurements leading to these values can be find in [35]. . . . .	28
4	Datasets types and model input description. . . . .	31
5	Training models parameters and test metrics of models trained based on random simulated samples of a given size from detected reactions. . . . .	34

# 1 Introduction

In the Baltic Sea, mostly in the Gotland Deep and in the Bornholm Deep, during warfare, over 250 kilotons of munition were sunken. Most of this arsenal consists of explosives and chemical agents. Besides official dumping sites, there is a lot of sunken munition in unknown locations [1][2].

The current approach to the detection of underwater threats is based mostly on sonars. This technique, which uses sound propagation, provides shapes of investigated objects but does not have the capability to identify substances, which the objects consist of [3]. Thus, in many cases, in order to make a decision if an inspected object is hazardous or not, one needs to perform an additional check, which is done by specially trained divers, which put their health and lives at risk. A novel, non-invasive device designed to remotely detect hazardous substances in the aquatic environment using neutron activation analysis is being developed at the Jagiellonian University within the framework of the SABAT (Stoichiometry Analysis By Activation Techniques) project.

Neutron Activation Analysis (NAA) is an analytical, non-destructive method that uses neutron beams to determine the concentrations of elements in the unknown substance. During neutron irradiation elements in the investigated sample capture neutrons and become radioactive isotopes. As a result, a range of particles may be emitted from the excited nuclei, including neutrons, protons, and different kinds of radiations, such as alpha, beta, or gamma [4]. The emitted gamma quanta are characteristic for a certain isotope. Knowing that hazardous materials mainly consist of carbon, oxygen, hydrogen, and nitrogen, the detection of gamma quanta allows to determine if a given substance is dangerous [1][2]. A schematic representation of principles of the NAA method is presented in Figure 1, where a nucleus is excited by inelastic scattering of a neutron and emits a gamma ray which then may be detected.

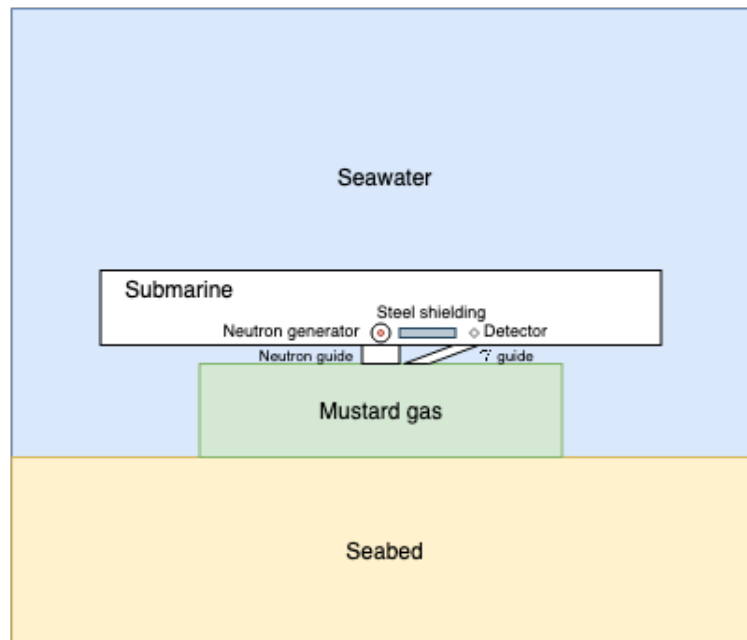


**Figure 1:** A schematic representation of principles of the NAA method. A neutron scatters inelastically with an atomic nucleus leading to its excitation. Transition to the ground state of the nucleus leads to gamma ray emission which is registered by a detector [2].

The SABAT system aims to detect hazardous materials in the underwater environment based on Neutron Activation Analysis. Simulations of the SABAT device have been already performed.



These have been carried out using Monte Carlo methods and primarily have focused only on the detection of mustard gas. The simulated system geometry is shown in Figure 2. A white rectangle represents a submarine. It is made of 3 mm stainless steel and has dimensions of 300 x 300 x 40 cm<sup>3</sup>. It contains a neutron generator with the 14 MeV neutron source, shown as a red dot, and target represented as a black circle around it. Under the neutron generator, outside the submarine, there is a cuboid neutron guide (white rectangle) of size 20 x 20 x 10 cm<sup>3</sup>. In the submarine, there is also a detector, represented as a light-grey square, which has a cylindrical shape with both height and diagonal length of 2 inches. It is placed 50 cm from the neutron generator. Under the detector, there is a gamma quanta guide. It has a shape of a polyhedron with 20 x 20 cm<sup>2</sup> top base, 20x7.7 cm<sup>2</sup> bottom base, and height of 35 cm. Both neutron and gamma quanta guides are made from 3 mm thick stainless steel. On the seabed (yellow rectangle) of simulated size 400 x 400 x 100 cm<sup>3</sup> there is a 3 mm thick steel container (194 x 50 x 50 cm<sup>3</sup>), represented as a green rectangle, with mustard gas. There have been performed two series of simulations. In the first series, the mustard gas container is covered with 1 cm of wood, and the second series is without it. In the submarine and in both of the guide tubes there is air under normal pressure. Surrounding seawater, with a salinity of 7.8‰, contains trace elements described in [5] on page 12 in the last column of Table 1. The values, which were expressed in mg/kg, were normalized to the salinity of the water. The seabed is sandy sediment, which consists in 25% of water and contains trace elements specified in [3] on page 4 in Table 1. Simulated environment corresponds to the areas of the most interest, which is the bottom of the Baltic Sea near to the shores. Additionally, there have been carried out corresponding two series of simulations without mustard gas.



**Figure 2:** Scheme of the simulated SABAT system investigating mustard gas in aquatic environment [3].

The goal of the research presented in this thesis was to check feasibility of fast identification of a hazardous substance using neural networks. These studies were done based on the Monte Carlo simulations of the SABAT detector investigating a mustard gas container. Moreover, influence of a gamma quanta detector resolution on the precision of this new recognition method was checked. The main technologies used for this project were Python 3.7, Keras library, and Jupyter notebook.

This thesis is structured into 5 Chapters. In the second Chapter, an introduction to machine learning and a detailed explanation of neural networks are given. Chapter 3 contains information about simulated data from the SABAT detector. The essential part of this chapter is the description of data pre-processing, and generation of datasets to train, validate, and test models. These models to recognize mustard gas are presented together with corresponding datasets in Chapter 4. Chapter 5 contains conclusions and the outlook for the future. Lastly, in Chapter 6 there is a summary of the thesis.

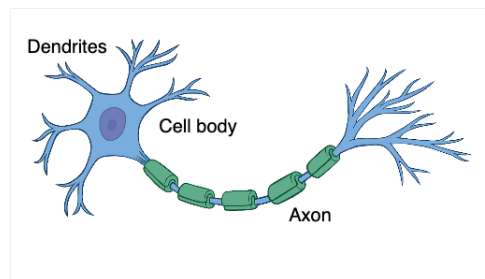
## 2 Neural Networks

Machine learning is a "field of study that gives computers the ability to learn without being explicitly programmed" [6]. It is a set of algorithms that gives the capability to learn from data. They can be classified into four groups: supervised, unsupervised, semi-supervised, and reinforcement learning. Supervised learning algorithms are the ones that gain knowledge on how to analyse objects based on labeled training data. The most obvious examples of supervised models are classifiers. On the other hand, unsupervised learning algorithms acquire knowledge from neither labeled nor categorized data. They find and describe the hidden structure of the given data. The typical unsupervised learning algorithm is clustering. Semi-supervised learning is somewhere in between two previously described categories since it uses both labeled and unlabeled data sets [7]. The reinforcement learning models are based on interactions with the environment. The model called agent performs actions and receives rewards or punishment based on which it is able to learn. One of the machine learning approaches is neural networks. The concept and the name of these mathematical structures refer to biological neural networks that can be found in brains. It was introduced for the first time in 1943 by mathematician Walter Pitts and neurophysiologist Warren McCulloch. They described the nervous system as a net of neurons. Each one of neurons has a threshold which excitation exceeding leads to the initialization of an impulse [8].

Research presented in this thesis is focused on neural networks for supervised learning.

### 2.1 Biological neuron

In the end of 19th century, spanish neuroscientist and pathologist Santiago Ramón y Cajal discovered that the brain and spinal cord consist of nerve cells called neurons. A neuron is formed of a cell body, dendrites, and an axon [9]. A basic structure of a nerve cell is shown in figure 3.



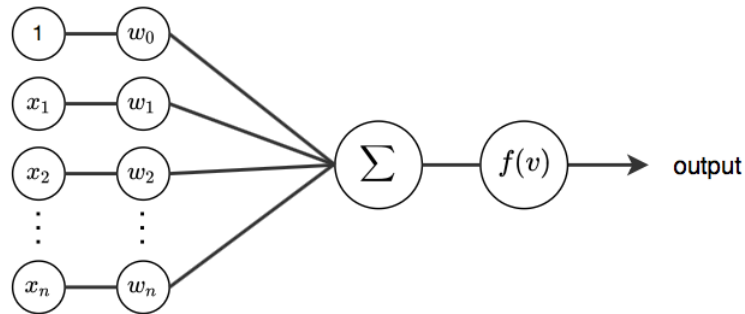
**Figure 3:** A schematic representation of a basic nerve cell. Adapted from [10].

The core part of a neuron is the cell body. It controls the cell's functions. Dendrites attached to the cell body allow its communication with the environment and with other neurons. The long tail-like structure, which is called an axon, extends the cell body and is responsible for carrying the signal away from a neuron. Neuron communicates through synapses in which dendrites of one neuron and axons of other neurons do not touch each other [10]. The space between them is called the synaptic gap. John Carew-Eccles' research showed that there are excitatory and inhibitory

synapses [11], which means that the signal may be increased or decreased between neurons with endogenous chemicals.

## 2.2 Mathematical model of neuron

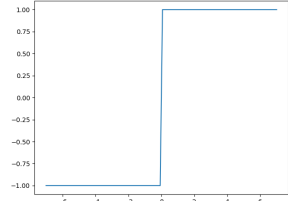
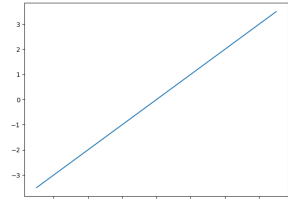
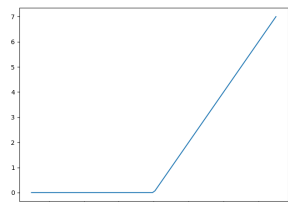
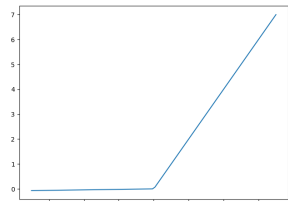
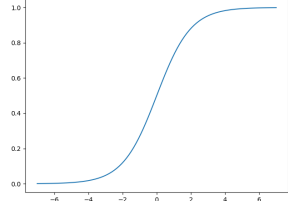
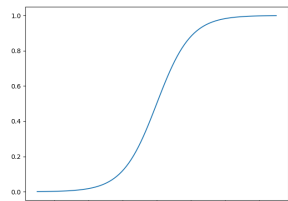
An artificial neuron is a simplified model of a biological one. Input vector  $x = (x_0, x_1, x_2, \dots, x_n)$  fulfills the role of neural signals registered by dendrites. It is assumed that  $x_0 = 1$  to reduce bias (defined in Section 2.5). Each input has a corresponding weight as in biological neuron synapses are able to influence the signal. Weights vector is represented by  $w = (w_0, w_1, w_2, \dots, w_n)$ . Inside neuron there is a net input function given as a weighed sum of inputs  $v = g(x, w) = w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \dots + w_n \cdot x_n = w^T \cdot x$  and an activation function  $y = f(v)$ . Since in the case of a real nerve cell there is only one axon, the artificial neuron model contains only one output  $y$  [9]. A scheme of the described artificial neuron is presented in Figure 4.



**Figure 4:** Scheme of a mathematical model of a neuron. It consists of input vector  $x = (x_0, x_1, x_2, \dots, x_n)$ , weights  $w = (w_0, w_1, w_2, \dots, w_n)$ , a net input function denoted as  $\Sigma$ , and an activation function  $f(v)$  [12].

### 2.2.1 Activation functions

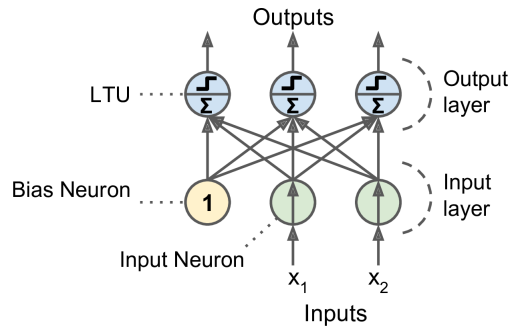
An activation function defines the outcome signal of a neuron. There are many functions, which can be used in neural networks for that purpose. In Table 1 the most popular ones are presented.

Name	Formula	Plot
Binary Step	$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases} \quad (1)$	
Linear	$f(x) = x \cdot m, \text{ where } m \in \mathbb{R} \quad (2)$	
ReLU	$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \quad (3)$	
Leaky ReLU	$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0,01 \cdot x & \text{if } x < 0 \end{cases} \quad (4)$	
Sigmoid	$f(x) = \frac{1}{1 + e^{-x}} \quad (5)$	
Hyperbolic Tangent	$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \quad (6)$	

**Table 1:** Selected activation functions used with supervised neural networks.

## 2.3 Perceptron

The perceptron, which was "invented in 1957 by Frank Rosenblatt", is one of the simplest artificial neural network architecture [13]. It contains one dense layer of neurons described in section 2.2 with binary step activation function, called linear threshold units (LTU) [13]. Each neuron is connected to all inputs. These connections are represented by input neurons, which pass thorough all information that was sent to them. There is also one additional input neuron that always outputs a value of 1. It is called a bias neuron. An example of a perceptron with 2 input and 3 output neurons is shown in Figure 5.



**Figure 5:** Schema of a perceptron consisting of 2 input and 3 output neurons. Adapted from [13].

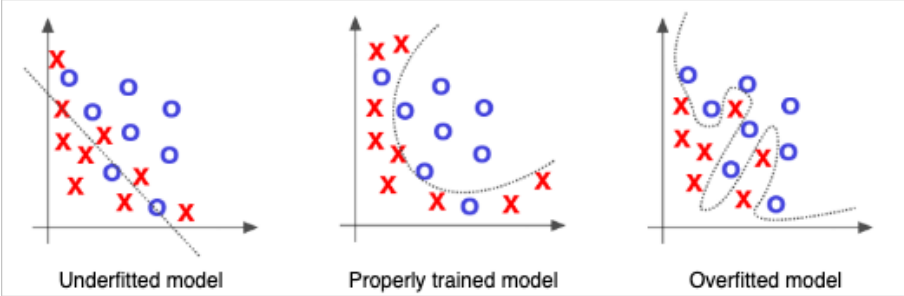
## 2.4 Feedforward Neural Network

There are many architectures of neural networks. They may differ in the degree of complexity and learning potential. One of them is the Feedforward Neural Network (FNN). Its name comes from the information flow, which is directed only forward within the network. In other types of networks, information may go backward or within one layer [14]. As in the case of perceptron, all layers are fully connected and each layer except the output layer contains bias neuron. Feedforward Neural Networks may contain additional layers of neurons, called hidden layers, between the input and output ones. Networks with more than one such a layer are known as deep neural networks [13]. Moreover, one defines layer width as the number of neurons contained in the layer (excluding bias neurons), and network depth which is the number of layers (without the input one) [15]. Furthermore, modern FNNs may contain additional techniques applied between or in dense layers. They are described in Section 2.5.

## 2.5 Neural network learning process

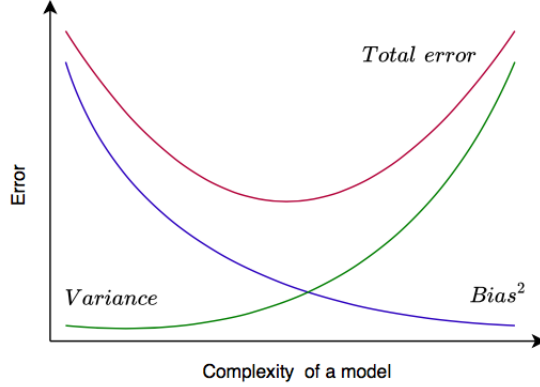
This section is devoted to the training of an FNN in the context of supervised learning. It is a process, in which a model is provided with input data, known as features and correct results - called labels. The aim of the training is to process the features - both known, from which the model learns, and previously not seen by the model, in a way to get the closest possible results to the real ones [14]. It can be achieved by manipulating trainable parameters - weights. Each neuron has a set of weights as described in section 2.2. While training the network, weights of all neurons

are being updated. The total number of weights in a network equals the number of links between neurons (including bias neurons). The number of trainable parameters has an impact on model capacity. Insufficient capacity may lead to underfitting. It occurs when a model cannot recognize the underlying pattern of the data. It performs poorly for both training and testing data sets. On the other hand, too great complexity may cause overfitting. It happens when a model recognizes not only the underlying pattern but includes also noise and outliers. An overfitted model performs very well for known data and poorly for unseen data points. Performance of a neural network in all the listed cases is pictorially presented in Figure 6.



**Figure 6:** Pictorial representation of the underfitted (left plot), overfitted (right plot) and properly trained neural networks. In the plots, the crosses and the circles represent data points of two different classes, and the dashed line illustrates a pattern of a model. Adapted from [16].

The complexity of a model is connected with variance and bias of its predictions. Variance is the variability of predictions for a given data points. High variance models focus on too many details on the training set and therefore they are overfitted. Bias is defined as a difference between correct values and an average prediction of the model. High bias models oversimplify the pattern by not paying enough attention to the training data set [17]. It is not possible to create a model with low variance and low bias at the same time. While the variance is decreasing, the bias is increasing. Finding an optimal model complexity is known as a variance-bias tradeoff (see Figure 7).



**Figure 7:** Qualitative dependence of the neural network performance as a function of the complexity. The optimum performance is achieved by minimization of the total error defined as the sum of variance, the bias squared and so called irreducible error, which is a measure of noise in the data [18].

An important role in the learning process is played by a loss function, also called a cost function, which describe the distance between the calculated results and the correct ones. The most commonly used loss functions for binary classification are described below. In all of these functions, the best possible result is a value of 0, and the greater it is, the worst the model behaves.

### 2.5.1 Binary cross-entropy

Binary cross-entropy, also known as negative log-likelihood, is a loss function intended for binary classification problems with labels values  $x \in \{0, 1\}$ . To obtain model results in the range  $[0, 1]$  the last layer of the network should have sigmoid as an activation function (see Section 2.2.1).

According to Shannon's theory of information, entropy is an average information. Information is given as a number of bits which is necessary to encode a random event from a given probability distribution. Entropy can be understood as "the average or expected uncertainty associated with this set of events" [19]. It is described by the equation:

$$H(X) = - \sum_{i=1}^n p(x_i) \cdot \log_b(p(x_i)) \quad (7)$$

Where:

- $X$ : is an array of events,
- $n$ : is a length of  $X$ ,
- $p(x_i)$ : is a probability of event  $x_i$ ,
- $b$ : is a base of the logarithm, commonly used  $b \in \{2, e, 10\}$ .

The binary cross-entropy, as the name indicates, is based on entropy. An event for each  $x_i$  in a dataset is defined as belonging or not to a given class A:



$$y_i = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases} \quad (8)$$

The probability of an event is simply the prediction of a model. That is, if an output of a network for a given element  $x_j$  is  $model(x_j) = \hat{y}_j$  then predicted probability of belonging to class A equals  $p(x_j \in A) = \hat{y}_j$  and predicted probability of element not being in class A is  $p(x_j \notin A) = 1 - \hat{y}_j$ . Binary cross-entropy is a function of two parameters - arrays of the same lengths. It is given by the equation:

$$BCE(Y, \hat{Y}) = -\frac{1}{n} \sum_{i=1}^n y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i) \quad (9)$$

Where:

- $Y$ : is an array of correct values,
- $\hat{Y}$ : is an array of predicted values,
- $n$ : is a length of labels arrays,  $n = |Y| = |\hat{Y}|$ ,
- $y_i$ : is a real probability of element  $x_i$  being in class A,  $y_i \in \{0, 1\}$ ,
- $\hat{y}_i$ : is a predicted probability of belonging to class A.

### 2.5.2 Hinge Loss

The hinge loss function was mainly developed to use with support vector machines. It works well for classification problems with labels values  $x \in \{-1, 1\}$  [21]. The model obtains results in that range, for example with a hyperbolic tangent function as activation of the output layer. The hinge loss function is defined as:

$$HL(Y, \hat{Y}) = \frac{1}{n} \sum_{i=1}^n \max(0, 1 - y_i \cdot \hat{y}_i) \quad (10)$$

Where:

- $Y$ : is an array of correct values,
- $\hat{Y}$ : is an array of predicted values,
- $n$ : is a length of labels arrays,  $n = |Y| = |\hat{Y}|$ ,
- $y_i$ : is an i-th real value,
- $\hat{y}_i$ : is a predicted i-th value.

### 2.5.3 Hinge squared Loss

There are many extensions of the hinge loss function. The most known one is the hinge squared loss. It is simply a squared score for each tuple of real label and prediction before calculation of mean [22]:

$$HSL(Y, \hat{Y}) = \frac{1}{n} \sum_{i=1}^n \max(0, 1 - y_i \cdot \hat{y}_i)^2 \quad (11)$$

Where:

- $Y$ : is an array of correct values,
- $\hat{Y}$ : is an array of predicted values,
- $n$ : is a length of labels arrays,  $n = |Y| = |\hat{Y}|$ ,
- $y_i$ : is an i-th real value,
- $\hat{y}_i$ : is a predicted i-th value.

To prevent overfitting of the neural network one applies so called regularization methods. In most of the cases one adds an extra term to the loss function which is penalizing it, as in the two most known regularization methods described below.

### 2.5.4 R1 regularization

In this regularization method parameters of a neural network are being shrunk towards zero. The term added to the loss function is given as:

$$L1 = \lambda \sum_{i=1}^n |w_i| \quad (12)$$

Where:

- $\lambda$ : is a coefficient, the higher it is, the more the loss function is penalized and the more the weights of the model are shrunk toward zero,
- $n$ : is a number of weights in the model,
- $w_i$ : is an i-th weight.

A model with the penalized absolute value of the weights often computes a subset of input features, because features with weights equal to zero are omitted. Consequently, it is simpler and easier to interpret. However, it does not have the ability to learn complex data patterns. L1 regularization has multiple, sparse solutions. In addition it is robust to outliers [23].

### 2.5.5 R2 regularization

R2 regularization penalty term is a sum of squared weights of the model:

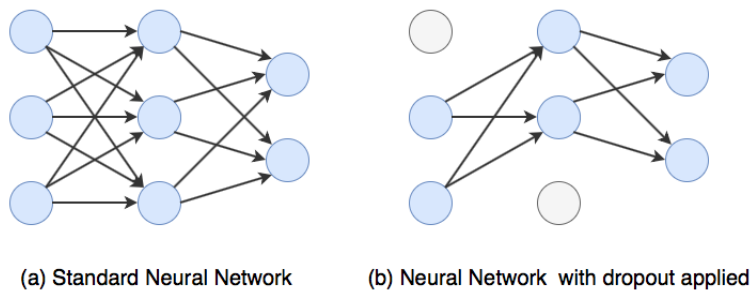
$$L2 = \lambda \sum_{i=1}^n w_i^2 \quad (13)$$

Where:

- $\lambda$ : is a coefficient, the higher it is, the more the loss function is penalized and the smaller are the weights,
- $n$ : is a number of weights in the model,
- $w_i$ : is an i-th weight.

Weights of the model with L2 regularization are small and not necessarily equal zero. The model is able to learn complex problems but it is not robust to outliers [23].

Dropout is another technique to obviate overfitting. At each training step, some neurons are being entirely ignored. Every neuron from the input layer and all hidden layers has a probability  $p$  of being "dropped out" at a given training step [13]. In Figure 8 an example of neural network without (a) and with dropout applied at the training stage is shown.



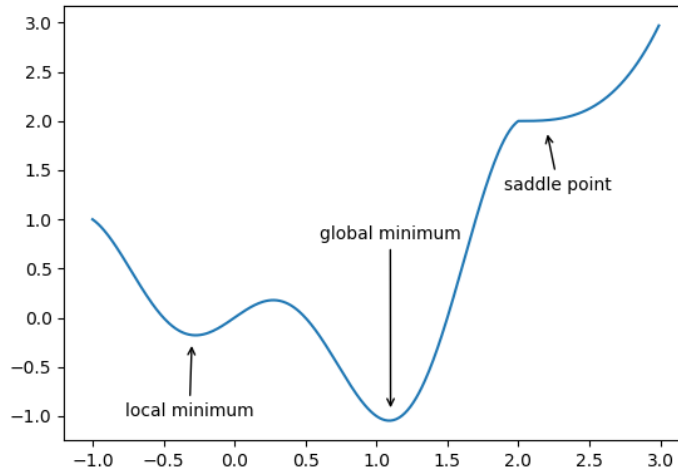
**Figure 8:** A standard neural network without dropout (a) and the neural network with two omitted neurons at the training stage with applied dropout (b) [24].

### 2.5.6 Optimization algorithms

There are many optimization problems in deep learning. The most important and at the same time most difficult one is training a model. This means tuning model parameters to significantly reduce the cost function. Optimization for training a machine learning task is different from pure optimization. The aim of the training is to minimize a generalization error, which is intractable. It could be simplified to minimizing an error with respect to the test set. Hence, the model parameters can only be optimized indirectly by reducing cost function with respect to the training set. Hopefully, while doing that, the test error and unknown generalization error will decrease as well. On the contrary, in pure optimization problem minimizing the training error would be the goal itself [25].

All of the commonly used optimization algorithms are gradient-based. Let's assume there is a one-dimensional loss function  $f : \mathbb{R} \rightarrow \mathbb{R}$ . The slope of  $f(x)$  at the point  $x$  equals the derivative  $f'(x)$ . It indicates how the output of the function will change corresponding to a small input modification:  $f(x + \epsilon) \approx f(x) + \epsilon \cdot f'(x)$  [25]. Therefore, it can be used in minimizing loss function to obtain a slightly better result in each iteration of an algorithm. Real-life loss functions are more complex, thus previously mentioned impact of input modification can be generalized to multivariate loss function  $g : \mathbb{R}^n \rightarrow \mathbb{R}$ , where the input is a vector  $x = [x_1, x_2, \dots, x_n]$ . Partial derivative  $\frac{\partial g(x)}{\partial x_k}$  for  $k \in [1, n]$  describes the change of output when only one parameter  $x_k$  is modified. The gradient  $\nabla g(x) = [\frac{\partial g(x)}{\partial x_1}, \frac{\partial g(x)}{\partial x_2}, \dots, \frac{\partial g(x)}{\partial x_n}]^T$  is a vector containing all partial derivatives [26].

Points where the derivative or all partial derivatives of loss function equal zero, contain no information about which direction to move in the next step. These critical points are local minima, local maxima, and saddle points. Once the algorithm steps into a point where the gradient is vanished it may not be able to overcome the mentioned critical point and find an optimal minimum. An example of a loss function with saddle point and local minimum is presented in Figure 9.



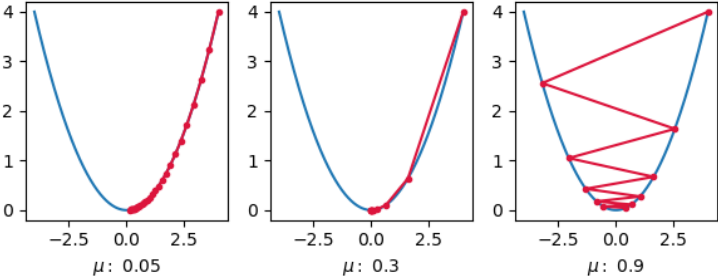
**Figure 9:** Example of a loss function with saddle point, local minimum, and global minimum [26].

Below the most common gradient-based optimization algorithms are presented. These optimizers focus only on training data, therefore they can find the best possible minima with respect to known data. Because of that, an optimization algorithm must be stopped at an optimal moment to avoid overfitting. All operations on vectors in sections 2.5.6.1 - 2.5.6.6 are element-wise.

### 2.5.6.1 Gradient Descent

Gradient Descent (GD) is a basic iterative optimization algorithm used to find a local minimum of a differentiable function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ , where  $n \geq 1$ . Let  $x_0$  be an initial point and let  $\mu$  be the learning rate. In each step a new point is defined as  $x_{i+1} = x_i - \mu \cdot \nabla f(x_i)$ . In this algorithm the gradient  $\nabla f(x)$  is calculated using the entire dataset, thus the cost of computing each iteration is  $\mathcal{O}(n)$ . In figure 10 there are shown usages of the algorithm with different learning rates on one-

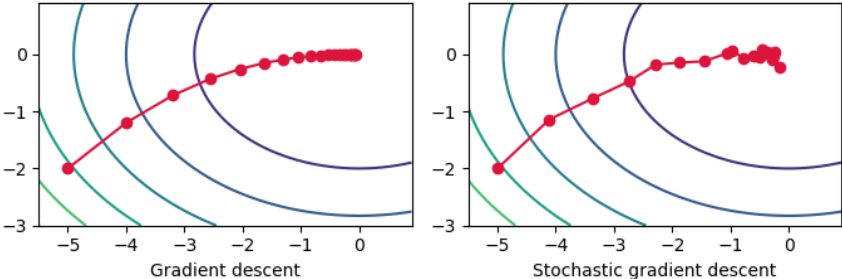
dimensional function. Choosing the correct learning rate can be difficult. Too small  $\mu$  may lead to too small steps and consequently not reaching a minimum. On the other hand, too big value of  $\mu$  could cause jumping out of the area. One of the approaches for this problem is to modify the learning rate during a model learning process.



**Figure 10:** Gradient descent with different learning rates: too small learning rate (left plot) require a lot of learning epochs before reaching a minimum, accurate learning rate (middle plot) enables to step into a minimum in optimal number of iterations, and too big learning rate (right plot) causes moving too far away in each of a step [26].

### 2.5.6.2 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a modification of the Gradient Descent algorithm where the gradient of a loss function is estimated based on a random fraction of a dataset. Because of this improvement, the computation cost of each iteration reduces from  $\mathcal{O}(n)$  to  $\mathcal{O}(1)$  in trade for insignificantly lower convergence rate. A comparison of GD and SGD steps for a two-dimensional function is shown in figure 11.



**Figure 11:** Comparison of Gradient Descent and Stochastic Gradient Descent convergence. The GD algorithm (left plot) in each iteration converges slightly better than the SGD algorithm (right plot). However, computation cost of each iteration of the SGD is much lower than the GD [26].

### 2.5.6.3 Momentum

The momentum method is a technique to accelerate GD or SGD by accumulating gradient vector in directions where reduction is persistent [27]. Let  $x_0$  be the initial point and  $v_0 = 0$  be the

variable to store gradients. In each step new point is defined by the following equations [28]:

$$v_{i+1} = \beta \cdot v_i + \nabla f(x_i) \quad (14)$$

$$x_{i+1} = x_i - \mu \cdot v_{i+1} \quad (15)$$

Where:

- $\beta$ : is the momentum coefficient,  $\beta \in [0, 1]$ ,
- $\mu$ : is a learning rate.

#### 2.5.6.4 AdaGrad

AdaGrad is a gradient-based algorithm with an adaptive learning rate. In each step, the gradient vector is being scaled down along the steepest dimensions. This means each training step is corrected towards the global optimum. Let  $x_0$  be the initial point and  $s_0 = 0$  be the parameter to accumulate gradients. New points are given by [13]:

$$s_{i+1} = s_i + (\nabla f(x_i))^2 \quad (16)$$

$$x_{i+1} = x_i - \frac{\mu}{\sqrt{s_{i+1} + \epsilon}} \nabla f(x_i) \quad (17)$$

Where:

- $\epsilon$ : is a constant greater than zero for numerical stability, typically set to  $10^{-10}$ ,
- $\mu$ : is a learning rate.

AdaGrad achieves good results for simple quadratic problems. However, the learning rate is often decreased too strongly and the algorithm cannot reach the global minimum when it is used for more complex patterns of the data [13].

#### 2.5.6.5 Root Mean Square Prop

Root Mean Square Prop (RMSProp) is a modification of AdaGrad algorithm which accumulates only the most recent gradients. It prevents from decreasing the learning rate too rapidly. At each training step a new point  $x_i$  and a parameter  $s_i$  ( $s_0 = 0$ ) are defined as [26]:

$$s_{i+1} = \gamma \cdot s_i + (1 - \gamma)(\nabla f(x_i))^2 \quad (18)$$

$$x_{i+1} = x_i - \frac{\mu}{\sqrt{s_{i+1} + \epsilon}} \nabla f(x_i) \quad (19)$$

Where:

- $\gamma$ : is the momentum coefficient,  $\gamma \in [0, 1]$ ,
- $\epsilon$ : is a constant greater than zero for numerical stability, typically set to  $10^{-6}$ ,
- $\mu$ : is a learning rate.

### 2.5.6.6 Adam

Adam, which stands for adaptive moment estimation, is an algorithm, which combines the advantages of both AdaGrad and RMSProp. Let  $x_0$  be an initial point,  $m_0 = 0$  the first moment vector and  $v_0 = 0$  the second moment vector. In each iteration a new point  $x_{i+1}$  is defined using the following equations [29]:

$$m_{i+1} = \beta_1 \cdot m_i + (1 - \beta_1)\nabla f(x_i) \quad (20)$$

$$v_{i+1} = \beta_2 \cdot v_i + (1 - \beta_2)(\nabla f(x_i))^2 \quad (21)$$

$$\widehat{m}_{i+1} = \frac{m_{i+1}}{1 - \beta_1^{i+1}} \quad (22)$$

$$\widehat{v}_{i+1} = \frac{v_{i+1}}{1 - \beta_2^{i+1}} \quad (23)$$

$$x_{i+1} = x_i - \frac{\mu \cdot \widehat{m}_{i+1}}{\sqrt{\widehat{v}_{i+1} + \epsilon}} \quad (24)$$

Where:

- $\beta_1, \beta_2$  : are hyperparameters, typically initialized to  $\beta_1 = 0.9$  and  $\beta_2 = 0.999$ ,  $\beta_k$  to the power  $i + 1$  is denoted as  $\beta_k^{i+1}$ ,
- $\epsilon$ : is a constant greater than zero for numerical stability, typically set to  $10^{-8}$ ,
- $\mu$ : is a learning rate, originally proposed value is  $\mu = 0.001$ .

### 2.5.7 Glorot and He weights initialization

The problem of vanishing or exploding gradients may be significantly alleviated by a proper weights initialization before training a network. There have been proposed Glorot initialization technique [30], known also as Xavier initialization, and its modification for ReLU (including its variants) activation functions called He initialization [31]. Each of these techniques was named after one of the authors. The weights are initialized by a random variable from a normal distribution with a mean of 0, and a standard deviation  $\sigma$ , or from a uniform distribution in the range  $(-r, r)$  as described in Table 2. The variables  $fan\_in$ ,  $fan\_out$  are respectively the number of incoming and outgoing connections for a neural network layer, which weights are being initialized.

	Normal distribution	Uniform distribution
Glorot initialization	$\sigma = \sqrt{\frac{2}{fan\_in + fan\_out}}$	$r = \sqrt{\frac{6}{fan\_in + fan\_out}}$
He initialization	$\sigma = \sqrt{2} \sqrt{\frac{2}{fan\_in + fan\_out}}$	$r = \sqrt{2} \sqrt{\frac{6}{fan\_in + fan\_out}}$

**Table 2:** Glorot and He initializations parameters for normal and uniform distribution, where the number of incoming and outgoing connections of a layer, which weights are being initialized are denoted as  $fan\_in$  and  $fan\_out$ , respectively [13].

### 2.5.8 Batch Normalization

Batch Normalization is a technique addressed to an *Internal Covariate Shift* problem. It refers to changing distribution of each layer's inputs, as the parameters of the previous layers are being changed during training [32]. Stabilizing the distributions of layers' inputs in a model may lead to reducing the problem of exploding/vanishing gradients and consequently allow to train the model in a more efficient way. In a model with batch normalization, there are extra operations added for each layer just before the activation functions. For each activation independently mean and variance of the inputs are calculated in the mini-batch  $B = \{x_1, x_2, \dots, x_m\}$  in order to normalize it and then scale and shift the results [13]. The algorithm is given by the equations:

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i \quad (25)$$

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \quad (26)$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad (27)$$

$$y_i = \gamma \hat{x}_i + \beta \equiv BN_{\gamma, \beta}(x_i) \quad (28)$$

Where:

- $m$ : is a number of training examples in the mini-batch  $B$ ,
- $\mu_B, \sigma_B$ : are the empirical mean and standard deviation calculated for the mini-batch  $B$ ,
- $\epsilon$ : is a constant greater than zero for numerical stability, typically set to  $10^{-3}$ ,
- $\hat{x}_i$ : is a normalized input,
- $\gamma, \beta$ : stand for the scaling and shifting parameters for the activation, respectively,
- $y_i$ : is the output.

Once the model has been trained the normalization is performed using population statistics with mean and variance averaged over mini-batches of size  $m$ . Consequently the transformation  $y = BN_{\gamma, \beta}(x)$  is replaced with the following definition of  $y$  [32]:

$$E[x] = E_B[\mu_B] \quad (29)$$

$$Var[x] = \frac{m}{m-1} E_B[\sigma_B^2] \quad (30)$$

$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} \quad (31)$$

$$y = \frac{\gamma}{\sqrt{Var[x] + \epsilon}} x + \left( \beta - \frac{\gamma E[x]}{\sqrt{Var[x] + \epsilon}} \right) \quad (32)$$

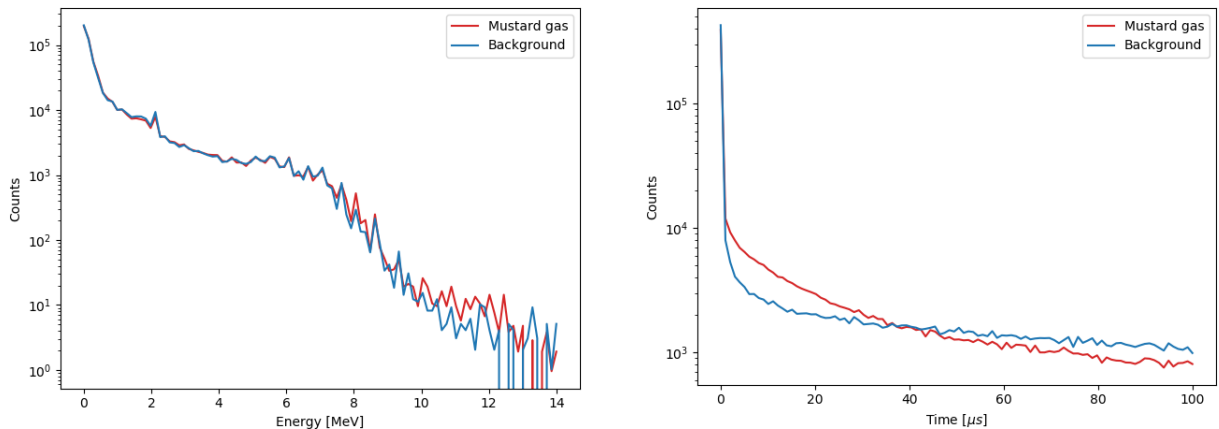


### 3 Data preparation

The SABAT Monte Carlo simulations contain information about each particle, which had interacted within the defined virtual world. It includes energy deposition, global time (counted since a neutron was emitted from a source), local time (counted since the particle was produced), coordinates of the reaction, particle name, kinetic energy, volume in which it had reacted, and other quantities. In order to prepare data sets relevant for the research presented in this thesis, only particles interacting in the detector were selected. For such subset of the simulated data only the energy deposited in the detector and the global time of interaction were taken into account, since only these quantities are provided in a real measurement. Thus, each reaction in the detector is represented by two values:

energy deposition ; global time

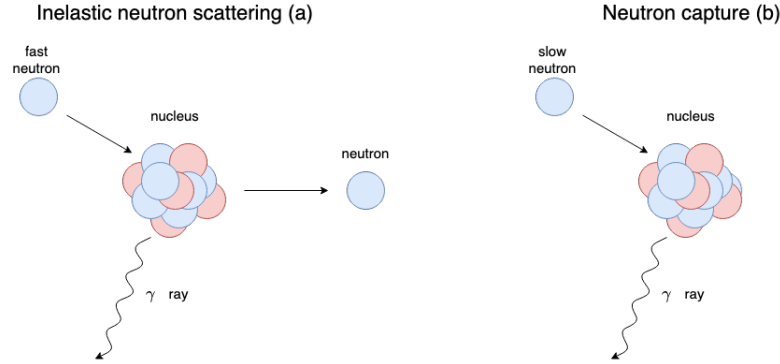
The data have been divided into two datasets. The first dataset contains information from simulations with mustard gas, surrounding water and sand, and is referred to as mustard gas or signal sample. In the second dataset, there are data from simulations without mustard gas which is referred to as background. Both mustard gas and background datasets contain around  $6 \cdot 10^5$  reactions. In Figure 12 spectra of energy deposition for both of the datasets are shown.



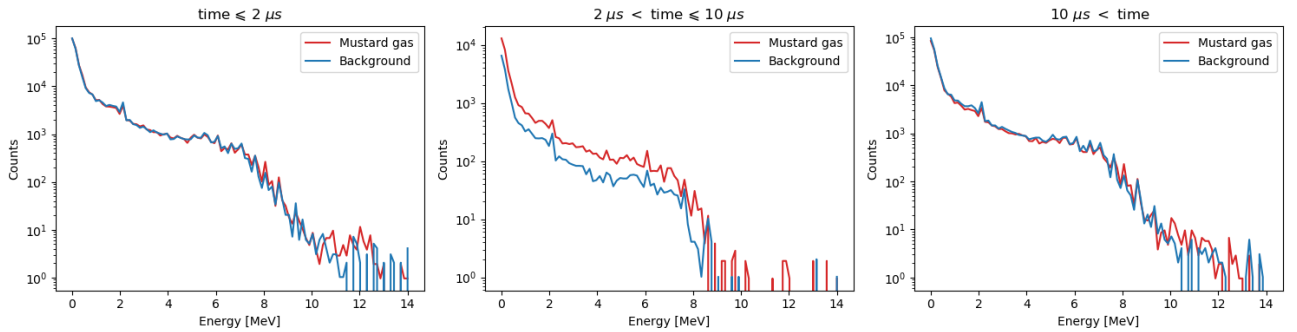
**Figure 12:** Energy deposition and global time in mustard gas and background datasets.

Interaction of a neutron with a nucleus which leads to emission of gamma quanta is an inelastic neutron scattering or neutron capture. Inelastic neutron scattering occurs when the speed of a neutron is high. During the reaction, the neutron is absorbed by a nucleus and then a neutron is re-emitted. Some energy of the neutron emitted from the source is absorbed by the recoiling nucleus and it remains excited. The excitation energy is released by emitting at least one gamma-ray [36]. This process is illustrated in Figure 13(a). On the other hand, if a neutron is slow, which means it had already lost most of its energy in other reactions, it may be captured by a nucleus it has collided with. As in inelastic neutron scattering after absorption of a neutron, the nucleus is

excited and emits one or more gamma rays, as presented in Figure 13(b). It has been estimated that in the case of underwater detection with the SABAT sensor, reactions which have occurred up to  $2 \mu s$  after a neutron was emitted are mainly inelastic neutron scattering. On the other hand, energy deposition detected in the detector after  $10 \mu s$  most likely originate from neutron capture [3]. Simulated distributions of energies deposited in the gamma quanta detector of the SABAT system are presented in Figure 14. The data were divided into three categories according to the time of registration of the gamma rays: less or equal to  $2 \mu s$ , greater than  $2 \mu s$  and less or equal to  $10 \mu s$ , and greater than  $10 \mu s$ .



**Figure 13:** Schema of inelastic scattering and neutron capture processes.

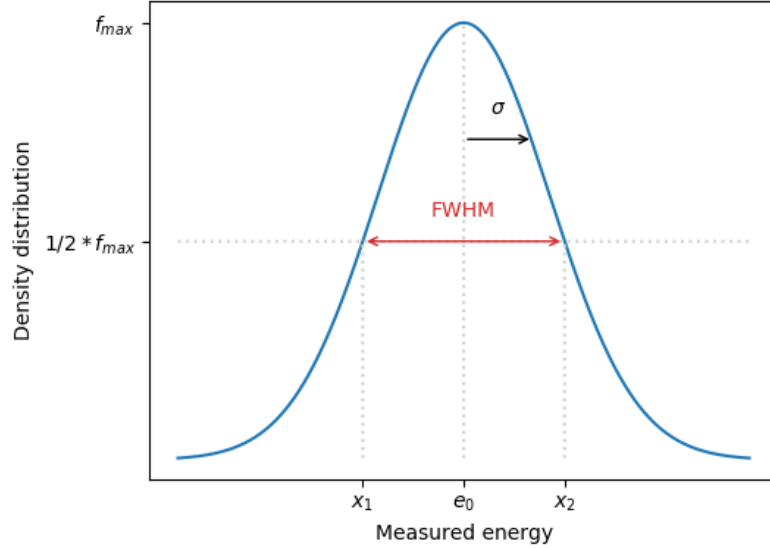


**Figure 14:** Simulated distributions of energies measured by the detector for three bins of gamma quanta registration time.

These spectra are done assuming a perfect detector time and energy resolutions which is not the case in the real measurement. Different detectors may differ in precision of determination of the energy of gamma rays. Energy resolution is defined as a full-width-at-half-maximum (FWHM) of a measured line divided by the true value of the gamma ray energy. FWHM is an absolute value of a difference between two extreme values, which fulfill the criterion  $f(x) = \frac{1}{2}f_{max}$ . It is illustrated in Figure 15.

Gamma quanta of energy  $e_0$  are registered by a detector with a finite resolution given by the  $\sigma$  parameter. The energy reconstructed by the detector may have different values according to the

Gaussian curve representing a probability density function. The sharper is the distribution peak the lower is its FWHM parameter, which means that subsequent measurements of the same energy line are concentrated more around the  $e_0$  value [33]. The relationship between FWHM and standard deviation  $\sigma$ , for a Gaussian-shaped curve, is defined as  $FWHM = 2\sigma\sqrt{2\ln 2} \approx 2.35\sigma$  [34].

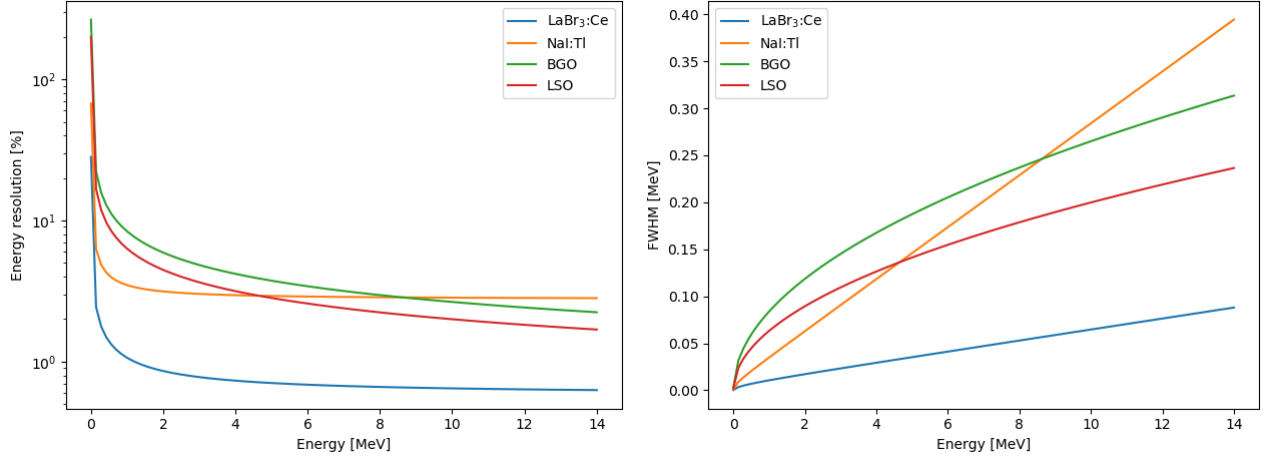


**Figure 15:** Full-width-at-half-maximum illustrated at a Gaussian-shaped peak [34].

Energy resolution  $R$  is energy dependent and can be parametrized using the following equation:  $R(e) = \sqrt{\frac{a^2}{e} + b^2}$ , where  $e$  is the gamma quanta energy in MeV, and  $a$ ,  $b$  are constant parameters [35]. Therefore, FWHM for a registered gamma quantum of energy  $e$  can be parametrized as:

$$FWHM(e) = \frac{R \cdot e}{100} = \frac{\sqrt{\frac{a^2}{e} + b^2} \cdot e}{100} \quad (33)$$

Parametrizations of the Full Width at Half Maximum dependence on gamma rays energy, according to Equation (33), are presented in Table 3 for most commonly used scintillation materials. The dependence of the resolutions of these materials as a function of energy are shown in Figure 16. As one can see detector offering the best measurement of the gamma rays energy is the  $\text{LaBr}_3:\text{Ce}$  but at the same time it is the most expensive scintillator. On the other hand, the cheapest material ( $\text{NaI:Tl}$ ) provides much worst performance of the detector. Thus, studies described in this work were done assuming usage of all the listed scintillating detectors in order to check if the proposed method of data analysis allows to use lower quality and cheap materials like  $\text{NaI:Tl}$ .



**Figure 16:** Energy resolution [%] and FWHM [MeV] for detectors.

Detector	a	b
Lanthanum Bromide (LaBr <sub>3</sub> :Ce)	0.893	0.582
Sodium Iodide (NaI:Tl)	2.131	2.759
Bismuth Germanate Oxide (BGO)	8.378	0
Lutetium Oxyorthosilicate (LSO)	6.319	0

**Table 3:** Energy resolution coefficients for most commonly used scintillating detectors. The details of measurements leading to these values can be find in [35].

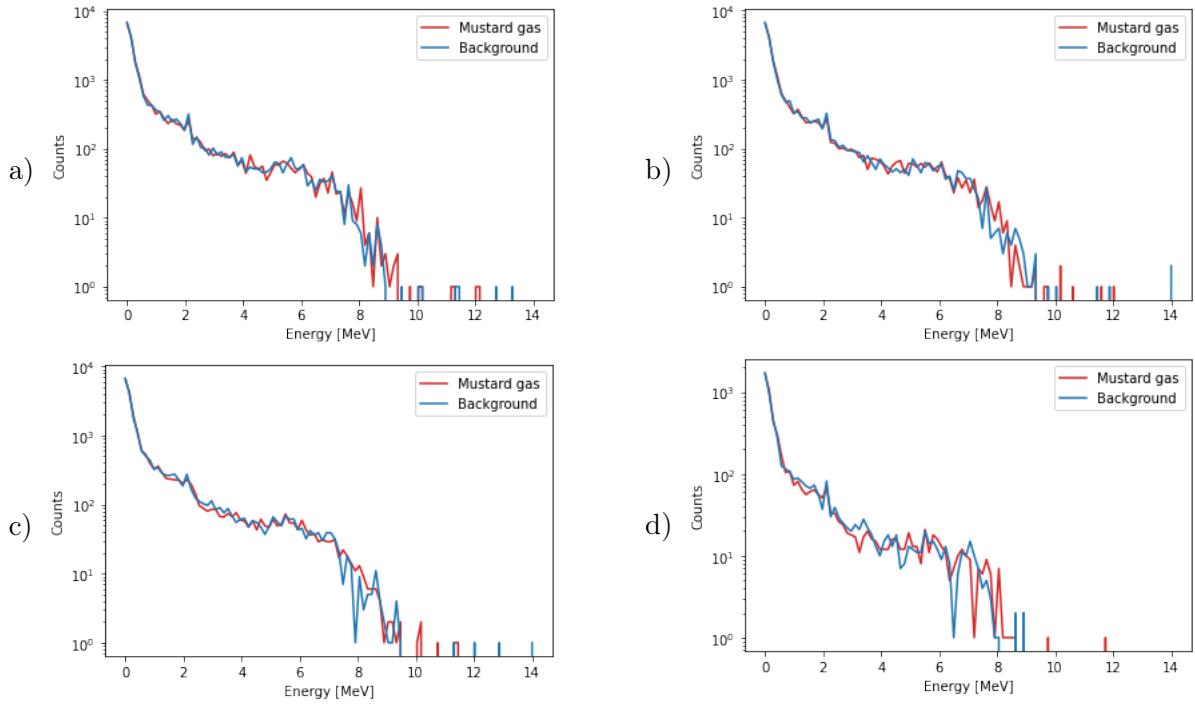
In order to train and validate models k-fold cross validation was used. The preferred quantity in both testing and validation sets is 15% of all data. The integer, which gives the closes size to 15% after division into folds is 7. In order to perform 7-fold cross-validation, both signal and background datasets have been randomly divided into 7 sets. Then, for each of the energy resolution parametrizations described in Equation (33) and Table 3 the simulated energy depositions were smeared with a Gaussian function. There have been created separate datasets containing tuples of energy and time, where each energy value  $e$  have been replaced by a random number from Gaussian distribution with mean  $\mu = e$  and standard deviation  $\sigma = \frac{FWHM(e)}{2\sqrt{2\ln 2}}$ . Such a smeared simulation data were used as an input to the training of the neural network models described in this work. Data samples were composed from randomly chosen events represented as histograms of energy deposition done for the three time intervals defined earlier in this section. Example data samples are shown in Figure 17 and 18. All the input data sets used to train the neural network and test its performance are explained in detail in Table 4 in the next chapter. Based on events in each of the 7 folds  $5 \cdot 10^3$  data samples were created. Thus, the total number of data samples for signal and background is  $7 * 10^4$ . In the first iteration of 7-fold cross-validation the first fold of both signal and background is used as a test set. One randomly chosen fold of the remaining 6 sets is used as validation data

and the rest served as a training set. Consequently, in the following 6 iterations the algorithm was trained with datasets constructed in the same way, having as the test set  $i$ -th fold, where  $i \in [2, 7]$  is the number of iteration. Additionally, in each of the iterations, the data were standardized, using the following equation:

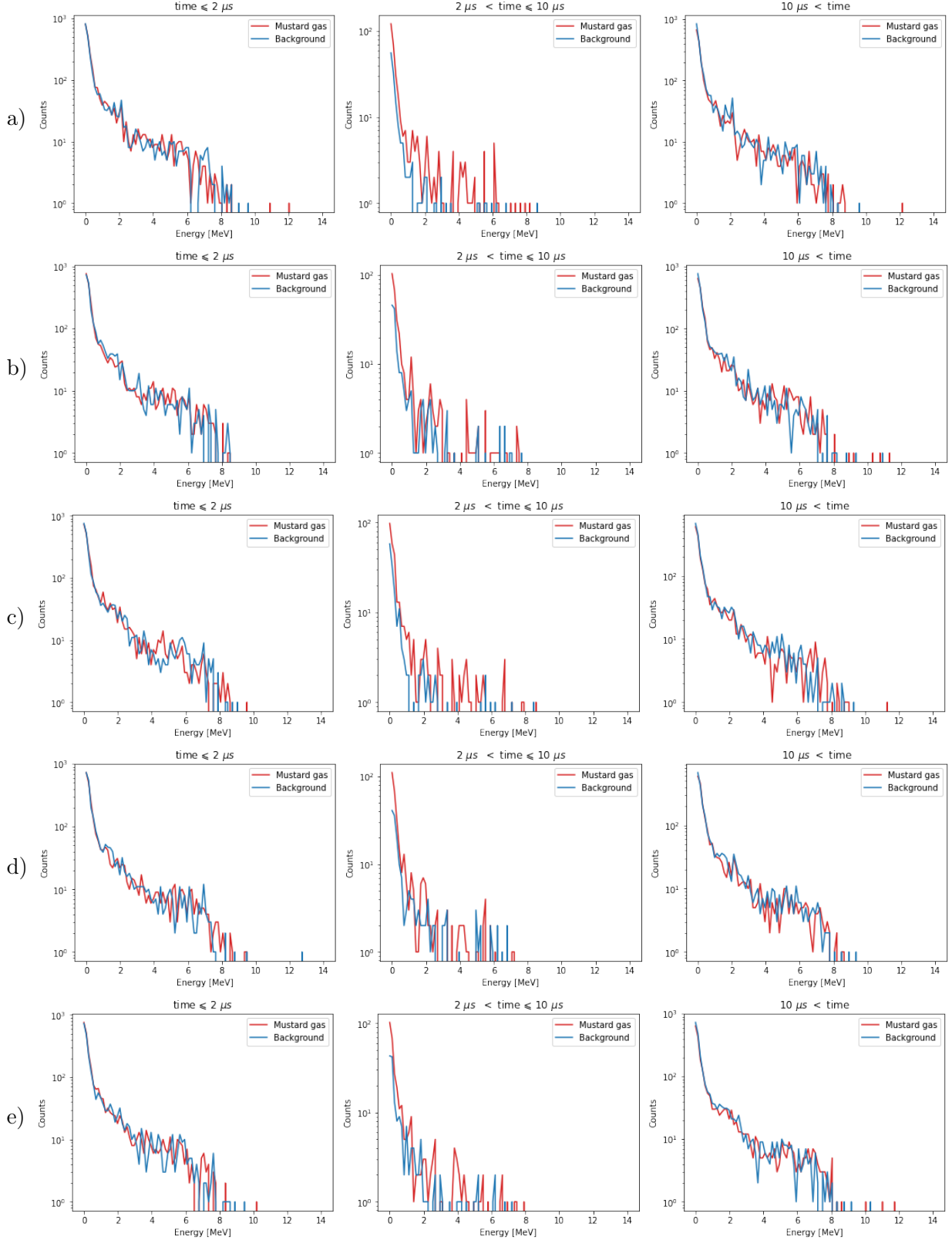
$$\hat{x}_{ij} = \frac{x_{ij} - \mu_i}{\sigma_i} \quad (34)$$

Where:

- $\hat{x}_{ij}$ : is a standardized  $i$ -th input value of  $j$ -th data point,
- $x_{ij}$ : is an  $i$ -th input value of  $j$ -th data point,
- $\mu_i$ : is a mean value of  $i$ -th input value, calculated over training data set,
- $\sigma_i$ : is a standard deviation of  $i$ -th input value, calculated over training data set.



**Figure 17:** Exemplary simulated data samples based on randomly chosen events and represented as histograms of energy deposition: a) without taking into account gamma quanta detector resolution, sample of  $2 \cdot 10^4$  events, b) with energy resolution corresponding to the LaBr<sub>3</sub>:Ce detector ( $2 \cdot 10^4$  events), c) applying Gaussian smearing using parametrization for the NaI:Tl detector ( $2 \cdot 10^4$  events), d) without taking into account gamma quanta detector resolution ( $5 \cdot 10^3$  events).



**Figure 18:** Example data samples based on  $5 \cdot 10^3$  randomly chosen simulated events represented as histograms of the energy deposition for three slots of the gamma quanta registration time. Histograms were done without energy smearing ( a ) and applying energy resolution parametrization for b)  $\text{LaBr}_3:\text{Ce}$ , c)  $\text{NaI:Tl}$ , d)  $\text{BGO}$  and e)  $\text{LSO}$  materials.

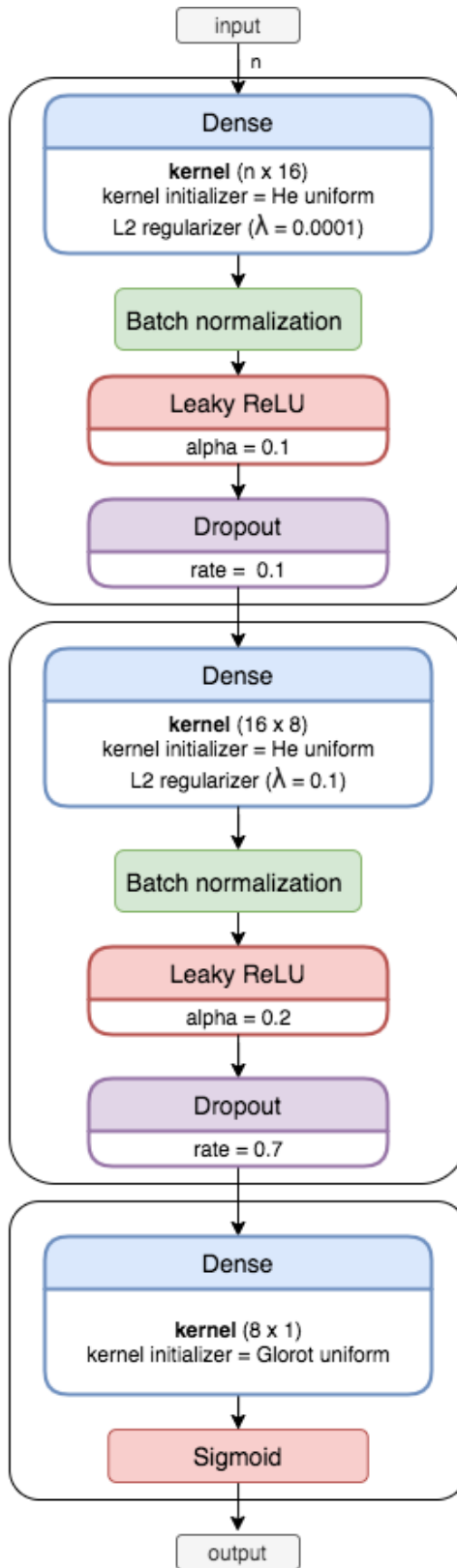
## 4 Trained models and their performance

In order to train models on different data sets a base model architecture was created. Its schema is shown in Figure 19. This model can be configured using the variable  $n$ , which is a width of the input data, depending on the dataset  $n \in \{100, 200, 300\}$ . It consists of 2 hidden layers, which contain respectively 16 and 8 units. For each of the hidden layers, we have used L2 regularization, with coefficient  $\lambda = 0.0001$  and  $\lambda = 0.1$ , respectively. After each of the described layers there is a batch normalization layer. Then we have used a leaky ReLU activation function with  $\alpha = 0.1$  and  $\alpha = 0.2$ , respectively. The last part of hidden layers blocks were dropout layers with rate equal respectively to 0.1 and 0.7. After the last hidden layer block, there was an output layer, which contained one unit followed by a sigmoid activation function. Hidden layers weights were initialized using He uniform initializer while the output layer was initialized with Glorot uniform initializer.

Models were trained on six types of datasets described in Table 4. A model type is associated with an input width  $n$ . The types are labeled using letters A - F for simplification purposes.

Dataset type	n	Sample representation
A	100	A histogram of all energy depositions with 100 bins.
B	300	Three histograms of energy depositions. The first 100 numbers represent the first histogram (reactions, which occurred within 2 $\mu s$ ), the following 100 numbers represent the histogram of energies detected after 2 $\mu s$ and before or exact at 10 $\mu s$ . The last 100 numbers represent the third histogram, which contains reactions detected after 10 $\mu s$ . The time of reaction is counted since the neutron was released.
C	200	Two histograms of energy depositions. The first histogram (reactions within 2 $\mu s$ ) is represented on the first 100 numbers, and the next 100 numbers represent histogram made of reactions, which occurred after 2 $\mu s$ since the neutron was released.
D	100	A histogram of energy depositions, which occurred within 2 $\mu s$ since the neutron was released.
E	100	A histogram of energy depositions, which occurred after 2 $\mu s$ and before or exact at 10 $\mu s$ since the neutron was released.
F	100	A histogram of energy depositions, which occurred after 10 $\mu s$ since the neutron was released.

**Table 4:** Datasets types and model input description.



**Figure 19:** Architecture of the neural network chosen to be used as classifier for the SABAT data analysis.



We have trained four models based on histograms of all energies, not taking into account the time of the reactions (dataset type A). First of all, there have been trained models on histograms, with sample size equal to  $2 \cdot 10^4$  smeared according to the energy resolution of detector  $\text{LaBr}_3:\text{Ce}$ , which is expected to be the most accurate, and for less precise, inexpensive  $\text{NaI}:\text{Tl}$  detector. It has also been performed on not smeared SABAT simulations for comparison purposes. As it could be expected, better results were achieved for model based on data from the  $\text{LaBr}_3:\text{Ce}$  detector, with a mean error rate from all 7 iterations of cross-validation equal to 0.00219. It is 0.00086 higher than the mean error rate obtained on the data without taking into account the energy resolution, and 0.00535 lower than  $\text{NaI} : \text{Tl}$  models. It turned out that the sensitivity of a detector has a great impact on a model’s precision when the input of a model is a histogram of energies disregarding the time of the reactions. Furthermore, we have checked the influence of the data sample size on the performance of the trained neural network. All the used algorithms were trained with  $5 * 10^3$  random samples of simulations without taking into account the detector resolution. The obtained mean accuracy was found to be equal to 0.93326 which shows that the size of the training samples is significant for this kind of methods.

Moreover, we have trained models, which predict based on three histograms, regarding the time of the reactions (dataset type B,  $5 * 10^3$  samples) for all the simulated detectors and the not smeared data. Ideal accuracy equal to 1.0 in all of the 7 iterations of cross-validation for  $\text{LaBr}_3:\text{Ce}$  model, with a mean loss of 0.00632. The model working on not smeared data obtained slightly better mean loss, which amounts to 0.00603. In a model based on BGO energy parametrization data samples mean accuracy insignificantly deteriorates into 0.99999. Models for detectors LSO and  $\text{NaI}:\text{Tl}$  performed equivalently with mean accuracy of 0.99993, and mean loss equal to 0.00383 and 0.00538, respectively. Additionally, in order to determine, which of the analyzed histograms, and hence, which of the reactions, inelastic neutron scattering or neutron capture, contain more information, we have prepared models with data samples from original data with datasets types D, E and F (see Table 4). The experiment has shown, that neutron capture reactions carry more information about investigated material, then inelastic scattering. Secondly, reactions, which occur between  $2 \mu\text{s}$  and  $10 \mu\text{s}$  contain a significant part of the information, due to differences in the number of detections in this time slot for the mustard gas and the background.

All of the described models used Adam optimizer and binary cross-entropy loss function. In table 5 training parameters such as batch size, learning rate, and a number of epochs are presented for each of the models. There one can find also means and standard deviations of accuracy and loss for all the models in 7-fold cross-validation iterations.

	Dataset type	Sample size	Batch size	Learning rate	Epochs	Mean accuracy	$\sigma_{Accuracy}$	Mean loss	$\sigma_{Loss}$
Original data	A	20000	750	0.0001	100	0.99867	0.00155	0.01737	0.00867
	A	5000	1000	0.0001	100	0.93326	0.03035	0.24739	0.24519
	B	5000	750	0.0001	100	1.00000	0.00000	0.00603	0.00122
	C	5000	1000	0.00004	160	0.94194	0.01868	0.18973	0.03627
	D	5000	1000	0.00005	70	0.80564	0.03427	0.83522	0.65892
	E	5000	750	0.0001	100	0.99989	0.00024	0.00501	0.00412
<i>LaBr<sub>3</sub> : Ce</i>	F	5000	750	0.0001	100	0.99341	0.00473	0.02886	0.01129
	A	20000	750	0.0001	100	0.99781	0.00260	0.02456	0.01244
	B	5000	750	0.0001	100	1.00000	0.00000	0.00632	0.00322
	A	2000	750	0.0001	100	0.99246	0.00708	0.03745	0.02450
<i>NaI : Tl</i>	B	5000	750	0.0001	100	0.99993	0.00013	0.00936	0.00538
	B	5000	750	0.0001	100	0.99999	0.00003	0.00929	0.00556
<i>BGO</i>	B	5000	1000	0.0001	100	0.99993	0.00017	0.00642	0.00383

**Table 5:** Training models parameters and test metrics of models trained based on random simulated samples of a given size from detected reactions.

## 5 Conclusions

This research has shown a great potential of mustard gas recognition based on spectra of energy deposition. Although the models had been trained on simulated data with limited size, effectiveness of signal recognition has been unexpectedly high. Taking into consideration that samples may be correlated in folds, training a chosen model should be repeated on real data measured with a chosen detector.

When a model is trained based on energy deposition, without taking into account the time of gamma quantum registration the sensitivity of a detector is crucial. In this case, Lanthanum Bromide ( $\text{LaBr}_3:\text{Ce}$ ) detector should be chosen. Moreover, the results indicate, that samples size is significant. It could be assumed, that increasing sample size largely, having unlimited data from a real detector, may improve the accuracy which have been obtained with  $\text{LaBr}_3:\text{Ce}$  detector and with samples size equal to  $2 \cdot 10^4$  events.

On the other hand, when a model is trained with the simulated data divided into three registration time slots, the sensitivity of a detector is insignificant. Ideal accuracy had been achieved on data from simulated  $\text{LaBr}_3:\text{Ce}$  detector with samples size equal to  $5 \cdot 10^3$  events. However, the accuracy of a model based on data from simulated Sodium Iodide ( $\text{NaI:Tl}$ ) detector, differs only by 0.00007. There is a high probability, that increasing dataset size and reducing correlation between samples by training a model on real data, may improve the results. Furthermore, it has been observed, that detected neutron capture reactions contain more information about the material, which is under investigation, than reactions of inelastic scattering. Moreover, differences in the number of detections in the time frame from  $2 \mu\text{s}$  to  $10 \mu\text{s}$  have been identified to carry an enormous amount of information.

## 6 Summary

During warfare in the Baltic Sea, there was sunken plenty of munition, including explosives and chemical agents. Unfortunately, the location of some sunken munition is still unknown. There have been proposed a novel device for non-invasive threats detection based on neutron activation analysis within the framework of the SABAT project at the Jagiellonian University. Simulations of the SABAT device which have focused primarily on the detection of mustard gas, have been carried out using Monte Carlo methods. Firstly, in a simulated environment, there has been mustard gas in a thick steel container on the sea bottom. Secondly, there have been performed corresponding simulations without the gas for background estimation.

The simulated SABAT device energy deposition histograms have been smeared with Gauss function with FWHM according to parametrization of the energy resolution for the most common scintillating detectors. Based on randomly chosen data samples we have created histograms of energy deposition for three slots of the time elapsed from the neutron generation to the registration of gamma quantum. The first histogram contained reactions, which have occurred within  $2 \mu s$ . Energy detected in this time interval mainly originated from inelastic neutron scattering. The second histogram consisted of energies detected after  $2 \mu s$  and before or exactly at  $10 \mu s$ . And the last histogram included reaction after  $10 \mu s$  which should come from neutron capture.

The datasets have been divided into 7 folds, in order to train and test neural network models using 7-folds cross-validation.

The research revealed that models based on energy depositions divided into interaction time slots are more accurate and require smaller data sample, than model, in which input data do not contain information about the time. The best results have been achieved with a model trained on data from simulated  $\text{LaBr}_3:\text{Ce}$  detector, with ideal accuracy of 1.0 and binary cross-entropy loss equal 0.00632. A model trained on data from simulated inexpensive  $\text{NaI:Tl}$  detector obtained slightly lower accuracy, which equals to 0.99993, with a loss of 0.00936. It was concluded that increasing dataset size and reducing correlation between samples by providing the data from a not-simulated detector may improve the results for  $\text{NaI:Tl}$  detector. Additionally, the results reveal that it is possible to create a model based on energy depositions, without the knowledge of the time of reactions, nevertheless, samples size has to be significantly greater, and high sensitivity of a detector is crucial.

## References

- [1] M. Silarski, P. Sibczyński, Sz. Niedźwiecki, S. Sharma, J. Raj, P. Moskal *Underwater detection of dangerous substances: status the SABAT project*, Acta Phys.Polon. B48 (2017) 1675-1683.
- [2] M. Silarski *Hazardous Substance Detection in Water Environments using Neutron Beams: the SABAT Project*, Problems of Mechatronics. Armament, Aviation, Safety Engineering 10 (2019) 49-60.
- [3] P. Sibczyński, M. Silarski, O. Bezshyyko, V. Ivanyan, E. Kubicz, Sz. Niedźwiecki, P. Moskal, J. Raj, S. Sharma and O. Trofimiuk *Monte Carlo N-Particle simulations of an underwater chemical threats detection system using neutron activation analysis*, JINST 14 (2019) P09001.
- [4] A. R. Barron, *Physical Methods in Chemistry and Nano Science*. OpenStax CNX. 21.07.2020 <http://cnx.org/contents/ba27839d-5042-4a40-afcf-c0e6e39fb454@20.17>
- [5] D.A. Kulik, J. Harff *Physicochemical modeling of the Baltic Sea water-sediment column: I. Reference ion-association models of normative seawater and of Baltic sea brackish waters at salinities 1-40 , 1 bar total pressure and 0 to 30° C temperature (system Na-Mg-Ca-K-Sr-Li-Rb-Cl-S-C-Br-F-B-N-Si-P-H-O)*, Meereswissenschaftliche Berichte 6 (1993) 1.
- [6] A. L. Samuel *Some studies in machine learning using the game of checkers*, IBM Journal (1959) 211-229.
- [7] <https://expertsystem.com/machine-learning-definition>, accessed 3.11.2019.
- [8] W. S. McCulloch, W. Pitts *A logical calculus of the ideas immanent in nervous activity*, Bulletin of Mathematical Biology (1990) 99-115.
- [9] M. Flasiński *Wstęp do sztucznej inteligencji*, Wydawnictwo Naukowe PWN SA, Warszawa (2011) 159-161.
- [10] <https://www.thepartnershipineducation.com/resources/nervous-system>, accessed 3.11.2019.
- [11] J. C. Eccles, *The ionic mechanism of postsynaptic inhibition*, Nobel Lecture, December 11, 1943.
- [12] <https://www.simplilearn.com/what-is-perceptron-tutorial>, accessed 4.11.2019.
- [13] A. Géron *Hands-On Machine Learning with Scikit-Learn & TensorFlow*, O'Reilly (2017), 257-258, 282, 297-300, 304.
- [14] J. Von Grabe, *Potential of artificial neural networks to predict thermal sensation votes*. Applied Energy 161 (2016) 412-424.
- [15] Z. Lu, H. Pu, F. Wang, Z. Hu, L. Wang *The Expressive Power of Neural Networks: A View from the Width*, NIPS (2017).

- [16] <https://www.kaggle.com/rafjaa/dealing-with-very-small-datasets>, accessed 25.02.2020.
- [17] <https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229>, accessed 25.02.2020.
- [18] <http://scott.fortmann-roe.com/docs/BiasVariance.html>, accessed 20.07.2020
- [19] MIT 6.02 DRAFT Lecture Notes, *Information, Entropy, and the Motivation for Source Codes*, September 13, 2012.
- [20] <https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a>, accessed 23.02.2020.
- [21] <https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/>, accessed 24.02.2020.
- [22] <https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/squared-hinge>, accessed 24.02.2020.
- [23] <https://medium.com/datadriveninvestor/l1-l2-regularization-7f1b4fe948f2>, accessed 17.03.2020.
- [24] S. Wang, X. Wang, P. Zhao, W. Wen, D. Kaeli, P. Chin, X. Lin *Defensive Dropout for Hardening Deep Neural Networks under Adversarial Attacks*, arXiv:1809.05165v1, 13 Sep 2018.
- [25] I. Goodfellow, Y. Bengio, A. Courville *Deep learning*, MIT Press (2016), 81, 271-272
- [26] [https://d2l.ai/chapter\\_optimization](https://d2l.ai/chapter_optimization), accessed 14.03.2020
- [27] I. Sutskever, J. Martens, G. Dahl, G. Hinton *On the importance of initialization and momentum in deep learning*, In Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28 (ICML'13). JMLR.org, III-1139-III-1147.
- [28] <https://distill.pub/2017/momentum/>, accessed 16.03.2020
- [29] D. P. Kingma, J. Lei Ba *Adam: A method for stochastic optimization*, arXiv:1412.6980v8, 23 Jul 2015.
- [30] X. Glorot, Y. Bengio *Understanding the difficulty of training deep feedforward neural networks*, AISTATS, volume 9 of JMLR Proceedings (2010) 249-256.
- [31] K. He, X. Zhang, S. Ren, J. Sun *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, arXiv:1502.01852v1, 6 Feb 2015.
- [32] S. Ioffe, C. Szegedy *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, arXiv:1502.03167, 2 Mar 2015.

- [33] T. K. Gupta *Radiation, Ionization, and Detection in Nuclear Medicine*, Springer Science Business Media (2013) 242-243
- [34] D. Locci-Lopez, R. Zhang, A. Oyem, J. Castagna *The Multi-Scale Fourier Transform.*, SEG (2018) 4176-4180.
- [35] A. Miś, *Badanie własności wybranych materiałów scyntylacyjnych pod kątem zastosowania w identyfikacji substancji niebezpiecznych*. Praca licencjacka, Kraków: Uniwersytet Jagielloński, 2018
- [36] <https://www.nuclear-power.net/nuclear-power/reactor-physics/nuclear-engineering-fundamentals/neutron-nuclear-reactions/neutron-inelastic-scattering/>, accessed 04.06.2020.

## A Python code

### A.1 Data preparation

#### A.1.1 Gaussian smearing

```
1 import random
2 import math
3 import abc
4
5
6 class BaseFWHM(abc.ABC):
7     def __init__(self):
8         self.a = NotImplemented
9         self.b = NotImplemented
10
11     def get_FWHM(self, e: float):
12         """
13         Returns full-width-at-half-maximum value for an energy value
14         """
15         return self.get_R(e) * e / 100
16
17     def get_R(self, e):
18         """
19         Returns energy resolution value for en energy value
20         """
21         if self.a is NotImplemented or self.b is NotImplemented:
22             raise NotImplementedError
23         return math.sqrt((math.pow(self.a, 2) / e) + math.pow(self.b, 2))
24
25     def blur_data_with_gauss(self, data: [(float, float)]):
26         """
27         Blurs energy from data with gauss calculating sigma from FWHM parameter.
28         """
29         blurred = []
30         for energy, time in data:
31             sigma = self.get_FWHM(energy) / 2.35
32             blur = random.gauss(0, sigma)
33             blurred.append((energy + blur, time))
34
35     return blurred
```



**Listing 1:** Base full-width-at-half-maximum

```
1 from src.gaussian_blur.base_FWHM import BaseFWHM
2
3
4 class LaBr3Ce_FWHM(BaseFWHM):
5
6     def __init__(self):
7         BaseFWHM.__init__(self)
8         self.a = 0.893
9         self.b = 0.582
```

**Listing 2:** Full-width-at-half-maximum of LaBr<sub>3</sub>:Ce detector

```
1 from src.gaussian_blur.base_FWHM import BaseFWHM
2
3
4 class NaITl_FWHM(BaseFWHM):
5
6     def __init__(self):
7         BaseFWHM.__init__(self)
8         self.a = 2.131
9         self.b = 2.759
```

**Listing 3:** Full-width-at-half-maximum of NaI:Tl detector

```
1 from src.gaussian_blur.base_FWHM import BaseFWHM
2
3
4 class BGO_FWHM(BaseFWHM):
5
6     def __init__(self):
7         BaseFWHM.__init__(self)
8         self.a = 8.378
9         self.b = 0
```

**Listing 4:** Full-width-at-half-maximum of BGO detector

```
1 from src.gaussian_blur.base_FWHM import BaseFWHM
2
3
```

```

4 class LSO_FWHM(BaseFWHM):
5
6     def __init__(self):
7         BaseFWHM.__init__(self)
8         self.a = 6.319
9         self.b = 0

```

Listing 5: Full-width-at-half-maximum of LSO detector

### A.1.2 Histograms generation

```

1 class DataSampling:
2
3     @staticmethod
4     def get_random_samples_without_replacement(data: list, n_samples: int, size: int):
5         """
6         Returns list of samples of a given size from the data. Elements in each sample
7         are unique, but they may be included in other samples.
8         """
9         samples = [random.sample(data, size) for _ in range(n_samples)]
10        return samples
11
12    @staticmethod
13    def get_random_samples_with_replacement(data, n_samples, size):
14        """
15        Returns list of samples from the data. Elements in samples may be not unique.
16
17        """
18        samples = [random.choices(data, k=size) for _ in range(n_samples)]
19        return samples

```

Listing 6: Data sampling

```

1 import sys
2 import numpy as np
3
4
5 class Edges:
6
7     @staticmethod
8     def get_time_edges(values):

```

```

9         """
10        Returns a list with edges, where first edge always equals zero
11        and last a maximum number.
12
13        """
14        return [0] + values + [sys.maxsize]
15
16    @staticmethod
17    def get_energy_edges(n, ranges, n_time_edges):
18        """
19        Returns the same bin edges for each time bins.
20        """
21        return [np.histogram_bin_edges([], n, ranges) for _ in range(n_time_edges + 1)]

```

Listing 7: Histogram edges

```

1 import numpy as np
2
3
4 class HistogramMaker:
5     """
6     2D Histogram with particles divided firstly by time and then by energy
7     """
8
9     @staticmethod
10    def make_histogram(data: [(float, float)], energy_edges: [list], time_edges: list):
11        """
12        Creates histogram of reactions with given time and energy edges.
13        The data should contain tuples with two values, where first value
14        is an energy and the second is time. Energy edges may be different
15        for each time bin.
16        """
17        divided = HistogramMaker._divide_energies_into_bins(data, time_edges)
18        hist = np.array([])
19        for times, edges in zip(divided, energy_edges):
20            hist = np.append(hist, np.histogram(times, bins=edges)[0])
21        return hist
22
23    @staticmethod
24    def _divide_energies_into_bins(data: [(float, float)], time_bins_edges: list):
25        """

```

```

26     Divide energies from data into n-1 lists based on it's time,
27     where n is a number of time edges.
28     """
29     divided = [[] for _ in range(len(time_bins_edges) - 1)]
30     for e, t in data:
31         for i, edge in enumerate(time_bins_edges):
32             if t <= edge:
33                 if i == 0:
34                     continue
35                 divided[i - 1].append(e)
36                 break
37
38     return divided

```

Listing 8: Histogram maker

```

1  import numpy as np
2
3  from src.histograms_generation.data_sampling import DataSampling
4  from src.histograms_generation.histogram_maker import HistogramMaker
5
6
7  def generate_histograms(histogram_maker: HistogramMaker, data: [(float, float)],
8                          n_samples: int, n_elements: int, energy_edges: [list],
9                          time_edges: [float], with_replacement=False):
10     """
11     Samples given number of samples with given number of elements and creates histogram
12     with given time and energy edges
13     """
14     if with_replacement:
15         samples = DataSampling.\
16             get_random_samples_with_replacement(data, n_samples, n_elements)
17     else:
18         if n_samples == 1 and n_elements == len(data):
19             samples = [data]
20         else:
21             samples = DataSampling.\
22                 get_random_samples_without_replacement(data, n_samples, n_elements)
23     histograms = []
24     for sample in samples:
25         histograms.append(histogram_maker.

```

```

26         make_histogram(sample, energy_edges, time_edges))
27
28     return np.array(histograms)

```

Listing 9: Generator

### A.1.3 Utils

```

1 class Loader:
2
3     @staticmethod
4     def load(path: str):
5         """
6         Loads data about particles from format: energy;time
7         """
8         file = open(path)
9         lines = file.readlines()
10        lines = [(lambda x: (float(x[0]), float(x[1].split('\n')[0])))(i.split(";")))
11                for i in lines]
12        return lines

```

Listing 10: Loader

```

1 import csv
2
3
4 class CsvSaver:
5
6     @staticmethod
7     def save_histograms(histograms: list, path: str):
8         """
9         Saves multiple histograms into csv file,
10        where each line represent one histogram.
11        Converts values to integers.
12        """
13        with open(path, 'w') as myfile:
14            wr = csv.writer(myfile, delimiter=';', quoting=csv.QUOTE_NONE)
15            for row in histograms:
16                row = [int(x) for x in row]
17                wr.writerow(row)
18

```

```

19
20     @staticmethod
21     def save_data(data: list, path: str):
22         """
23         Saves data into csv file
24         """
25         with open(path, 'w') as myfile:
26             wr = csv.writer(myfile, delimiter=';', quoting=csv.QUOTE_NONE)
27             for x in data:
28                 wr.writerow(x)

```

Listing 11: Saver

```

1 import math
2 import random
3
4 from src.loaders.basic_loader import Loader
5 from src.savers.csv_saver import CsvSaver
6
7
8 def _divide_into_k_folds(k: int, data_path: str, save_path: str, name: str):
9     # load data
10    loader = Loader()
11    data = loader.load(data_path)
12
13    # shuffle data
14    random.shuffle(data)
15
16    # divide into k folds
17    fold_length = math.ceil(len(data) / k)
18    folds = [data[offs:offs + fold_length] for offs in range(0, len(data), fold_length)]
19
20    for i, fold in enumerate(folds):
21        print(len(fold))
22        CsvSaver.save_data(fold, "{}/{}_{}.csv".format(save_path, name, i))

```

Listing 12: Dividing data into folds

## A.2 Training models

```

1 from keras import Sequential

```

```

2 from keras import regularizers
3 from keras.initializers import glorot_uniform, he_uniform
4 from keras.layers import Dense, LeakyReLU, BatchNormalization, Dropout
5
6
7 def get_model(input_shape):
8     model = Sequential()
9     model.add(Dense(16, input_shape=(input_shape,),
10                    kernel_regularizer=regularizers.l2(l=0.0001),
11                    kernel_initializer=he_uniform()))
12     model.add(BatchNormalization())
13     model.add(LeakyReLU(alpha=0.1))
14     model.add(Dropout(0.1))
15
16     model.add(Dense(8, kernel_regularizer=regularizers.l2(l=0.1),
17                    kernel_initializer=he_uniform()))
18     model.add(BatchNormalization())
19     model.add(LeakyReLU(alpha=0.2))
20     model.add(Dropout(0.7))
21
22     model.add(Dense(1, activation='sigmoid',
23                    kernel_initializer=glorot_uniform(), name='output'))
24     return model

```

Listing 13: Model

### A.2.1 Dataset preparation

```

1 import csv
2
3 import numpy as np
4
5
6 # Load data
7 def read_csv(filename):
8     """
9     Reads csv file with histograms.
10    The file should contain each histogram in separate line,
11    where counts in the bins are seperated by semi-colons.
12    """

```

```

13     with open(filename, newline='') as f:
14         reader = csv.reader(f, delimiter=';', quoting=csv.QUOTE_NONE)
15         histograms = [np.asarray([int(float(x)) for x in row])
16                       for row in list(reader) if '' not in row]
17
18     return histograms
19
20
21 def load_folds(dir_path, k):
22     """
23     Loads k folds for each of signal and background from files
24     named: signal_i, bcg_i where i is in 0..k
25
26     """
27     signal, bcg = [], []
28     for i in range(k):
29         signal.append(read_csv("{}signal_{}.csv".format(dir_path, i)))
30         bcg.append(read_csv("{}bcg_{}.csv".format(dir_path, i)))
31
32     return signal, bcg

```

Listing 14: Load folds

```

1 import numpy as np
2
3
4 def get_partial_data(folds, parts, part_lenght=100):
5     return [[_get_parts_of_list(x, parts, part_lenght) for x in fold]
6             for fold in folds]
7
8
9 def _get_parts_of_list(line, parts, part_lenght):
10    n_line = np.asarray([])
11    indices = [i * part_lenght for i in range(1, int(len(line) / part_lenght))]
12
13    slices = np.split(line, indices)
14    for part in parts:
15        n_line = np.concatenate((n_line, slices[part]))
16    return n_line
17
18

```



```

19 def get_data_with_summed_parts(folds, parts_to_sum, part_lenght=100):
20     return [[_sum_parts_of_list(x, parts_to_sum, part_lenght)
21             for x in fold] for fold in folds]
22
23
24 def _sum_parts_of_list(line, parts_to_sum, part_lenght):
25     n_parts = int(len(line) / part_lenght)
26     indices = [i * part_lenght for i in range(1, n_parts)]
27
28     slices = np.split(line, indices)
29     sum = np.zeros((part_lenght,))
30     for part in parts_to_sum:
31         sum = sum + slices[part]
32
33     n_line = np.asarray([])
34     new_parts = [x for x in list(range(n_parts)) if x not in parts_to_sum[1:]]
35     for x in new_parts:
36         if x != parts_to_sum[0]:
37             n_line = np.concatenate((n_line, slices[x]))
38         else:
39             n_line = np.concatenate((n_line, sum))
40
41     return n_line

```

Listing 15: Manipulation of input data

```

1 class Standardizer:
2     """ Standardizes data according to the data passed in initializer """
3
4     def __init__(self, X):
5         self.mean = np.mean(X, axis=0)
6         self.standard_deviation = np.std(X, axis=0)
7
8     def standardize(self, X):
9         standarized_X = np.array(
10             [(x - self.mean[i]) / (self.standard_deviation[i] + epsilon)
11              for i, x in enumerate(row)] for row in X)
12         return standarized_X

```

Listing 16: Standardizer

```

1 import numpy as np
2 import random
3
4
5 class CrossValidationDataProvider:
6
7     @staticmethod
8     def prepare_sets(folds_signal, folds_bcg):
9         sets = []
10
11         for i in range(k):
12             train_s, valid_s, test_s = CrossValidationDataProvider. \
13                 _divide_folds(i, folds_signal)
14             train_b, valid_b, test_b = CrossValidationDataProvider. \
15                 _divide_folds(i, folds_bcg)
16
17             x_train = train_s + train_b
18             y_train = [1] * len(train_s) + [0] * len(train_b)
19
20             x_valid = valid_s + valid_b
21             y_valid = [1] * len(valid_s) + [0] * len(valid_b)
22
23             x_test = test_s + test_b
24             y_test = [1] * len(test_s) + [0] * len(test_b)
25
26             s = Standardizer(x_train)
27             x_train = s.standardize(x_train)
28             x_valid = s.standardize(x_valid)
29             x_test = s.standardize(x_test)
30
31             sets.append({"train": (x_train, y_train), "valid": (x_valid, y_valid),
32                         "test": (x_test, y_test)})
33
34         return sets
35
36     @staticmethod
37     def _divide_folds(test_idx, folds):
38         ids = list(range(len(folds)))
39

```

```

40     test = folds[test_idx]
41     ids.remove(test_idx)
42
43     v = random.choice(ids)
44     valid = folds[v]
45     ids.remove(v)
46
47     train = []
48     for j in ids:
49         train = train + folds[j]
50
51     return train, valid, test

```

Listing 17: Cross validation data provider

## A.2.2 Visualization

```

1  import matplotlib.pyplot as plt
2
3  from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
4
5
6  def show_loss(model_hist, log=False):
7      n_epochs = len(model_hist.history['loss'])
8      epochs = range(0, n_epochs)
9      train_loss = model_hist.history['loss']
10     val_loss = model_hist.history['val_loss']
11
12     plt.plot(epochs, val_loss, '-.', label='Validation', color='tab:red')
13     plt.plot(epochs, train_loss, '-.', label='Training', color='tab:blue')
14     plt.xlabel('Epochs')
15     plt.ylabel('Loss')
16     plt.legend()
17     if log:
18         plt.yscale('log')
19     plt.show()
20
21
22  def show_accuracy(model_hist):
23     n_epochs = len(model_hist.history['accuracy'])

```

```

24     epochs = range(0, n_epochs)
25     train_loss = model_hist.history['accuracy']
26     val_loss = model_hist.history['val_accuracy']
27
28     plt.plot(epochs, val_loss, '-.', label='Validation', color='tab:red')
29     plt.plot(epochs, train_loss, '-.', label='Training', color='tab:blue')
30     plt.xlabel('Epochs')
31     plt.ylabel('Accuracy')
32     plt.legend()
33
34     plt.show()
35
36
37 def plot_confusion_matrix(predicted, y_test):
38     confusion_matrix = confusion_matrix(y_test, predicted.round(),
39                                       normalize='all')
40     _, ax = plt.subplots()
41     disp = ConfusionMatrixDisplay(confusion_matrix=confusion_matrix,
42                                  display_labels=["Background", "Mustard gas"])
43     disp.plot(include_values=True,
44              cmap='Blues', ax=ax, xticks_rotation='horizontal')
45     ax = disp.ax_
46     plt.grid(False)
47     ax.tick_params(labelsize=10)
48     plt.show()

```

**Listing 18:** Visualization of loss, accuracy and confusion matrix

### A.2.3 Trainig and testing models

```

1 import numpy as np
2 from keras.optimizers import Adam
3
4
5 def train_on_sets(sets, args):
6     histories = []
7     models = []
8     optim = Adam(lr=args['lr'])
9
10    for i, d_set in enumerate(sets):

```

```

11     print("Iteration: {}".format(i))
12     x_train, y_train = d_set['train']
13     model = get_model(input_length)
14     model.compile(optimizer=optim, loss='binary_crossentropy',
15                  metrics=['accuracy'])
16     history = model.fit(x_train, y_train, epochs=args['epochs'],
17                       batch_size=args['batch_size'],
18                       validation_data=d_set['valid'], shuffle=True)
19
20     models.append(model)
21     histories.append(history)
22
23     return models, histories
24
25
26 def show_histories(histories):
27     for i, history in enumerate(histories):
28         print("Iteration: {}".format(i))
29         show_loss(history, log=True)
30         show_accuracy(history)
31
32
33 def test_models(models, sets):
34     accuracies = []
35     losses = []
36
37     for i, model, d_set in zip(range(len(models)), models, sets):
38         print("Iteration: {}".format(i))
39         x_test, y_test = d_set['test']
40         accuracy, loss, predicted = get_test_metrics(model, x_test, y_test)
41         plot_confusion_matrix(predicted, y_test)
42
43         accuracies.append(accuracy)
44         losses.append(loss)
45
46     # calculate mean and std dev
47     mean_accuracy = np.mean(accuracies)
48     std_dev_accuracy = np.std(accuracies)
49     mean_loss = np.mean(losses)
50     std_dev_loss = np.std(losses)

```

```
51  
52     return mean_accuracy, std_dev_accuracy, mean_loss, std_dev_loss
```

**Listing 19:** Train and test models for cross validation