UNIVERSITÀ
DEGLI STUDI
DI PADOVA

UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Fisica e Astronomia "Galileo Galilei"

Corso di Laurea in Fisica

Tesi di Laurea

Discriminazione della forma d'impulso di segnali provenienti da scintillatori EJ-309 tramite tecniche di apprendimento automatico

Pulse Shape Discrimination of signals from EJ-309 scintillators via machine learning techniques

Relatore

Prof. Antonio Caciolli

Correlatore

Dr. Jakub Skowronski

Laureando

Edoardo D'Amore

Anno Accademico 2023/2024

Contents

Chapter 1

Abstract

Pulse Shape Discrimination techniques are used extensively for neutron/gamma identification when working with signals from scintillators. It is usual to have regions of the energy spectrum where these techniques are ineffective or show limitations in correctly separating signals coming from particles or gamma rays. In this work, we use multiple machine learning tools to improve neutron/gamma identification in such areas. We start by training an autoencoder model, from which we extract the features encoded in the latent layer. Next, we use several clustering algorithms, we compare them to find the one best suited for this task and then we train a Support Vector Machine to make predictions based on the data classified in this way.

Tecniche di disciminazione della forma d'impulso (PSD) sono usate estensivamente per l'identificazione di netroni o raggi gamma lavorando con segnali provenienti da scintillatori. Eúsuale avere regioni dello spettro energetico in cui queste tecniche sono poco efficaci o mostrano limitazioni nel separare correttamente segnali provenienti da particelle o raggi gamma. In questo lavoro utilizziamo molteplici metodi di apprendimento automatico per migliorare l'identificazione di netroni o gamma in quelle zone. Iniziamo allenando un modello di autoencoder, da cui estraiamo le feature codificate nel layer latente. Successivamente utilizziamo diversi algoritmi di clustering, li compariamo tra loro per trovare quello che meglio si addice al nostro problema e poi alleniamo una Support Vector Machine per effettuare predizione basate sui dati classificati in questo modo.

Chapter 2

Introduction

Detecting neutrons is extremely important in nuclear astrophysics and nuclear physics experiments. It can be done via neutron counters, as used in the study of the $^{13}C(\alpha, n)\,^{16}O$ reaction [2]. This method however doesn't allow for distinctions between the researched neutrons and those produced by impurities in the experimental setup. When working with fast neutrons, it is common to use scintillators to detect their presence and Pulse Shape Discrimination (PSD) techniques are the standard methods used when it comes to analyse their signals. Scintillators, however, are also able to detect gamma rays and, while PSD techniques are extremely efficient and effective in discriminating neutrons from gamma rays for most of the energy spectrum, they fail to do so correctly in the lower energy ranges. Neural networks can help overcome this limitation by finding unseen correlations in the data that better determine the signals nature.

Scintillators are light emitting materials often used when working with ionizing particles or radiation to detect their presence. They must satisfy certain characteristics. The first, and most important one, they must convert the particle's kinetic energy into light in a linear fashion, and must be transparent to the frequencies they emit. This process must be of prompt fluorescence, and not of phosphorescence or delayed fluorescence, as it is important to have short, pulse-like signals, results of a fast excitation and de-excitation of the material. Lastly, their index of refraction should be similar to that of glass [7].

Different materials have different characteristics: inorganic ones, like alkali halide cristals, generally have the best light output, both in amplitude and linearity, while organic ones, like naphtalene cristals, tend to have a much faster response time, although yelding less light [7]. This results in different use cases: the high Z-value of inorganic scintillators helps in the detection of gamma rays, while the hydrogen content of the organic ones makes them preferrable when working with beta radiation and fast neutrons.

2.1 Experimental setup

All the data used in this work comes from EJ-309 liquid organic scintillators that are part of a larger detection structure, called SHADES, located at Laboratori Nazionali del Gran Sasso (LNGS). It is composed of 12 EJ-309 scintillators and 18 ^{3}He counters [1]. These two types of detectors work together to discriminate between lower energy neutrons coming from the $^{22}Ne(\alpha, n)\,^{25}Mg$ reaction we are interested about, and higher energy ones coming from other background reactions, such as $^{10}B(\alpha, n)\,^{13}N$, $^{11}B(\alpha, n)\,^{14}N$ or $^{13}C(\alpha, n)\,^{16}O$. This type of scintillator is also able to detect the presence of gamma rays and so the main challenge we have to face is to correctly determine whether a signal is produced by a neutron or a gamma ray.

In figure 2.1 we can see two signals coming from one of the SHADES scintillators. All of our signals will have the same general shape: a sudden and high rise, followed by a rapid, but not instantaneus, decay. As the conversion from energy to light is linear, the particle's kinetic energy is proportional to

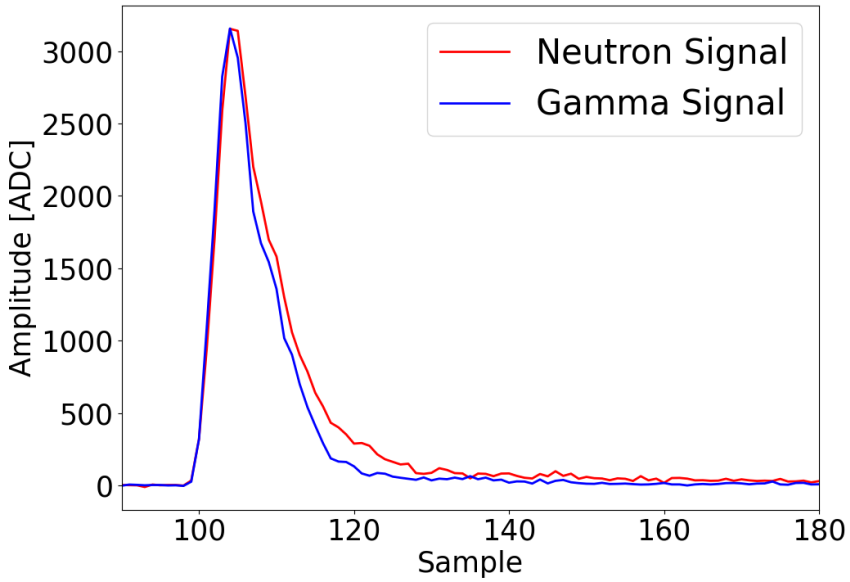

Figure 2.1: Two signals coming from a neutron and a gamma that have similar peak amplitude

the area under the signal curve. When working with this type of signals, it is common to use Pulse Shape Discrimination (PSD) techniques to better understand which particle has been detected. The one we used is Charge Integration (CI) and is based upon the fact that different type of particles produce a different ratio between the total signal area and just the signal tail area. In particular it is defined as:

$$\mathrm{CI} = \frac{Q_{\mathrm{long}} - Q_{\mathrm{short}}}{Q_{\mathrm{long}}} \tag{2.1}$$

where $Q_{\mathrm{short}} = \sum_{n=n_{\mathrm{rise}}}^{n_{\mathrm{peak}}} f_n \Delta t$, $Q_{\mathrm{long}} = \sum_{n=n_{\mathrm{rise}}}^{n_{\mathrm{tail}}} f_n \Delta t$, f_n is the n-th sample taken and $\Delta t = 2\mathrm{ns}$ is our sample rate [5]. In our case we have taken $n_{\mathrm{rise}} = 95$, $n_{\mathrm{peak}} = 110$ and $n_{\mathrm{tail}} = 160$. From now on we will refer to CI simply as PSD.

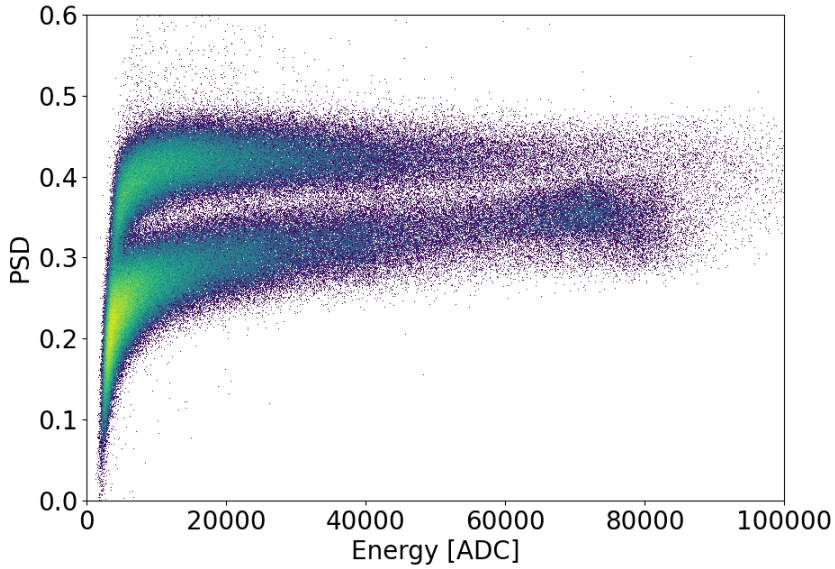

Figure 2.2: Plot of PSD vs energy for all of our signals

We can see the distribution of PSD over energy of our signals in figure 2.2. We can clearly identify two different clusters of data. As shown in figure 2.1, we can see that, on average, neutrons have a

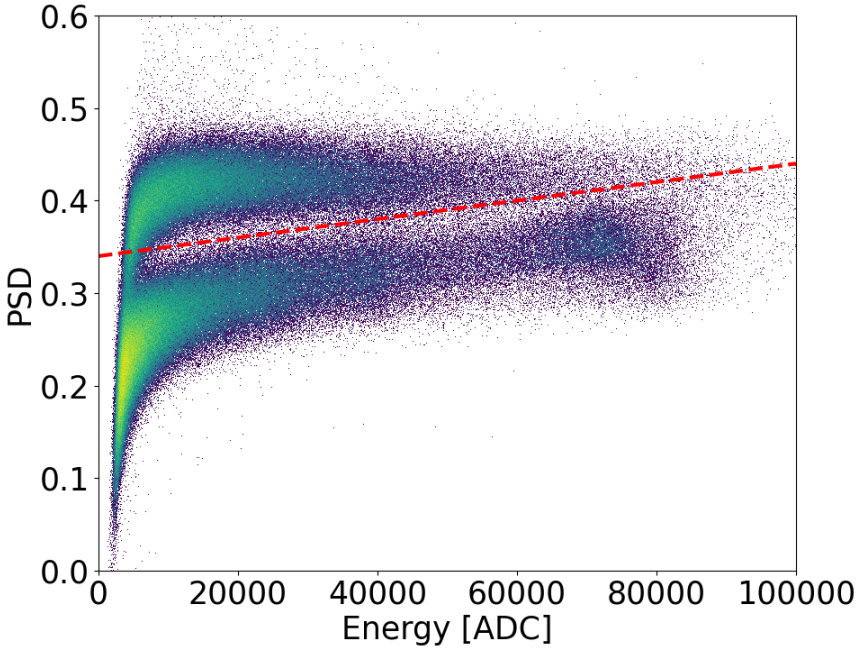

Figure 2.3: Example of a simple linear cut

bigger PSD value than gammas, as the signal tends to have a longer decay time, and so we can infer what the two groups represent and can easily separate them via a simple linear cut, just like in figure 2.3. While this method is very efficient and works well with most of the energy spectrum, it becomes quite problematic when applied to the lower energy areas as the two groups tend to merge and become indistinguishable from each other. This area is also very dense as about 37.4% of our signals have energy lower than $E_0 = 8000$ ADC.

To overcome this problem we decided to use a type of machine learning algorithms, called neural networks, to help us find correlations in the data that can not be found using traditional methods, such as the one we discussed above.

Chapter 3

Analysis

Artificial neural networks are, as the name implies, a set of individual units, called neurons, arranged and connected between themselves. Artificial neurons are loosely inspired by biological ones, as they receive various inputs from other neurons they are attached to, and output a signal that is generally the result of a function applied to the inputs. Basically every artificial neuron can be described as such:

$$o = a(\boldsymbol{w}^T \cdot \boldsymbol{x} + b) \tag{3.1}$$

where $\boldsymbol{x}$ is the input vector, $\boldsymbol{w}$ is the set of weights, b is a bias term and o is the output of the activation function a. These functions are non-linear to avoid the network functioning as a simple linear combination of the inputs and they should generally be differentiable almost everywhere. The first ever artificial neuron, the *perceptron*, had a simple step funcion as an activation function. Over the years many more have been tried and implemented, such as the sigmoid, the tanh, and the Rectified Linear Unit (ReLU), just to name some of the most impactful ones.

In a neural network, neurons are organized in layers. Each layer consist of neurons with the same inputs and the same activation function:

$$\boldsymbol{o}_l = a(\boldsymbol{W}_l^T \cdot \boldsymbol{x}_l + \boldsymbol{b}_l) \tag{3.2}$$

where $\boldsymbol{x}_l = \boldsymbol{o}_{l-1}$, $\boldsymbol{W}_l = (\boldsymbol{w}_{l,1}, \ldots, \boldsymbol{w}_{l,n})$, $\boldsymbol{b}_l = (b_{l,1}, \ldots, b_{l,n})$ for layer $l \in [0, L]$ with n neurons. We can observe how the inputs of layer l, and of each of his neurons, are basically the outputs of layer $l-1$. This defines a fully connected feed-forward network. Fully connected refers to the fact that each neuron of each layer is connected to all neurons from the previous and the next layers, while feed-forward describes how the information flows in an unperturbated forward manner from the input layer to the output layer [4]. We can see an example of the architecture of a simple network in figure 3.1, where the neurons are represented as circles and the connections are visualized via arrows emerging from them.

Apart from the first and the last layer, respectively called input and output layer, all the others are called hidden layers, as they are not as easily accessible as the other two and, for deep networks, we don't really know exactly what they are doing or what each neuron represents. What makes neural networks powerful is the ability of approximate nearly every type of function if given enough resources.

Being machine learning algorithms, they can be trained to "learn" the optimal weight values. There are three main approaches when it comes to learning: supervised, unsupervised and reinforcement. Depending on what problem we need to solve, we might opt for one or another. We will focus on the first two. Supervised learning consist of providing the network both the input $\boldsymbol{x}$ and the associated output $y = f(\boldsymbol{x})$ during training. During the training iterations, the network has to minimize a certain function $\mathcal{L}(o, y)$, called loss function, that depends on the difference between the provided output and the one the network came up with. At the end of the training the network should approximate the function f. Unsupervised training, as the name suggests, consists of providing the network with only the inputs $\boldsymbol{x}$. This results in the network figuring out by itself what are the relationships in the data.

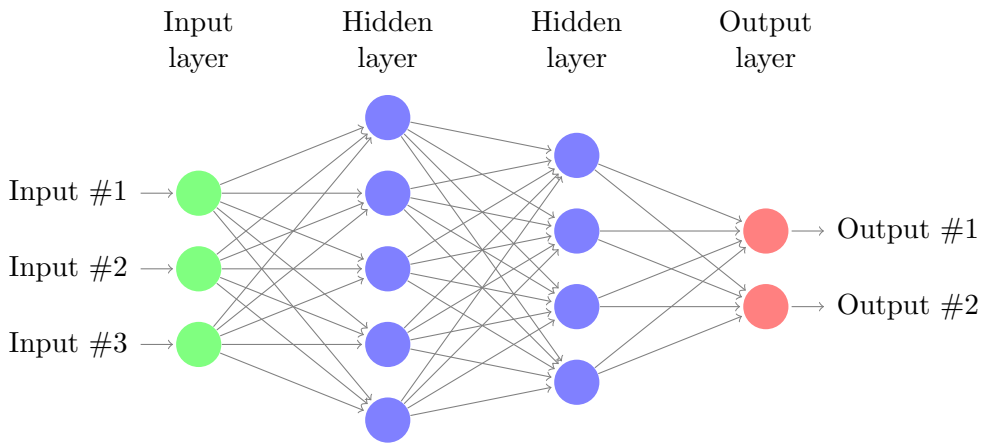

Figure 3.1: Representation of a basic neural network with 2 hidden layers

Regarding our problem, there have been studies where researchers have used supervised approaches, specifically convolutional neural networks, to improve the neuron/gamma discrimination, and obtained promising results [5]. The main issue in using a supervised method is the needing of classifying data beforehand, and, as we have seen, this could be impossible in low energy ranges. Even if possible, this classification could conceal certain biases that would be assimilated by the network. To avoid these problems we decided to try the unsupervised approach.

3.1 Building and training the autoencoder

An autoencoder is a neural network that tries to copy its input to its output. It is composed of two smaller networks: the encoder, which performs $\boldsymbol{h} = f(\boldsymbol{x})$, and the decoder, which performs $\boldsymbol{o} = g(\boldsymbol{h})$. The whole autoencoder is trained to minimize the difference between the input $\boldsymbol{x}$ and it's output $\boldsymbol{o} = g(f(\boldsymbol{x}))$. The three main layers we have to consider are the input $\boldsymbol{x}$, the output $\boldsymbol{o}$ and the hidden central layer $\boldsymbol{h}$, which we will call also latent layer, as it encodes the latent features of our data.

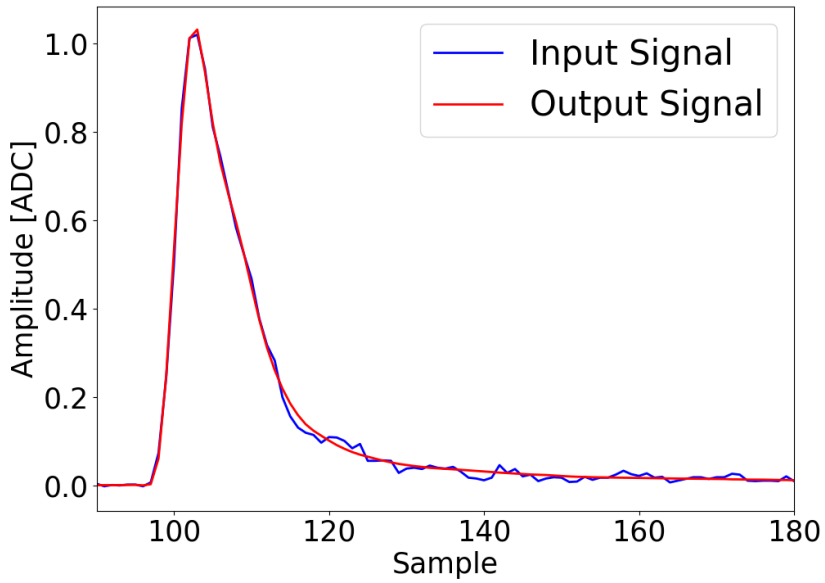

Figure 3.2: Plot of a signal before and after being processed by the autoencoder

We have used a specific type of autoencoder, called undercomplete autoencoder, where $\dim(\boldsymbol{h}) < \dim(\boldsymbol{x})$ [4]. This is reflected in the architecture as having less neurons in the latent layer $\boldsymbol{h}$ than the input layer $\boldsymbol{x}$. The reasoning behind this decision is that the network, to reproduce as accurately as

possible it's input, must encapsulate its most salient features in the latent layer, while discarding all the information that isn't essential to the reconstruction. When working with signal-like data, as we can see in figure 3.2, this results in the whole network functioning as a denoiser.

In figure 3.3 we show an example of a simple autoencoder architecture. We can see how, starting from the input, the dimentionality is progressively reduced until the latent layer, and then it is brought back in a specular fashion by the decoder. This double funnel symmetrical structure is pretty standard when working with this type of networks. The idea is that the decoder should function as an inverted encoder, to get $g(\cdot) \approx f^{-1}(\cdot)$. The number of neurons and layers of our autoencoder (without considering the bias terms) can be seen in table 3.1. The input and output layers size corresponds to the lenght of the signals. We wanted our latent layer to have as few neurons as possible to push the network into encoding physical variables that describe well our phenomena: energy, Q_{short} and Q_{long}. We found that having just 4 neurons was the best choice as just having 3 wasn't enough to properly discriminate and with 5 the network didn't perform as well in our task. After this imposition, all other network hyperparameters, such as the number of layers, their size and the activation functions, were manually fine-tuned to optimize our model. In particular, regarding the activation funtions, we compared several of them and decided to use an Exponential Linear Unit (ELU), instead of the more classic Rectified Linear Unit (ReLU) or the leaky-ReLU, as, out of the three, it led to a much better separation between the two groups when looking at the latent layer values over the whole dataset [3]. We considered also using a Scaled Exponential Linear Unit (SELU) but it differed negligibly from ELU.

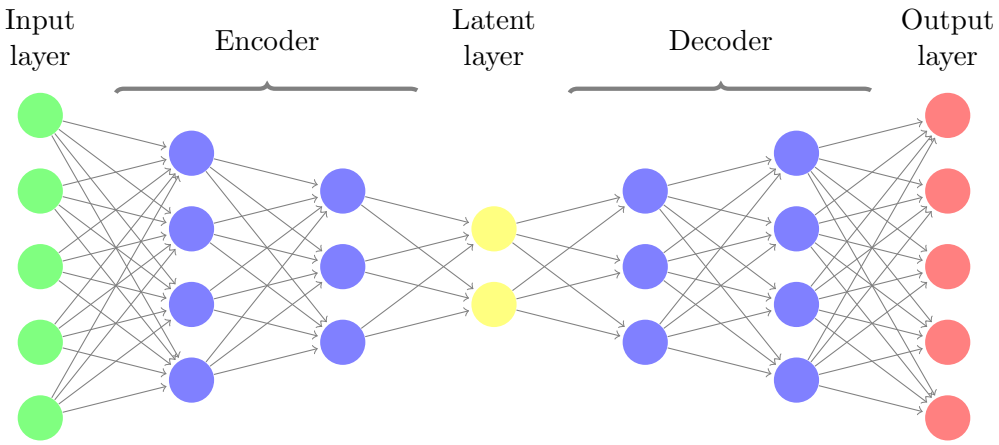

Figure 3.3: Representation of an autoencoder neural network

Layer	# of neurons
Input	872
Hidden 1	512
Hidden 2	256
Hidden 3	128
Hidden 4	64
Hidden 5	32
Latent	4
Hidden 6	32
Hidden 7	64
Hidden 8	128
Hidden 9	256
Hidden 10	512
Output	872

Table 3.1: Number of neurons in every layer of our autoencoder architecture

Before starting the training we must first choose what loss function to use. Given the nature of our signals, we decided to stick with a basic MSE (Mean Square Error) loss. The simplicity of this function allows us to easily interpret it's results. Another essential step is to select which optimizer to use. The optimizer is basically the algorithm that adjusts the weights of the network after each iteration of the training based on how was the performance, computed via the loss function. We chose the Adam optimizer as it is regarded as one of the best and most used optimizers and it is generally a good choice when there isn't a specific solution right away [6]. We decided to use a linearly decreasing learning rate (from lr = 0.1 to lr = 0.0001) over 100 epochs to boost our training in the first stages, allowing us to get as close to the mininum as possible in very few iterations, while leaving room for more fine-tuning during the later stages of the training.

The dataset we are working on is composed of almost 10^6 signals. When training machine learning models, it is good practice to divide the dataset into smaller separate sets: the training dataset, which, as the name suggests, is the one the model is actually trained on, the validation dataset, used during training to check if there are problems, such as overfitting, and the testing dataset, used after the training has finished to further validate how well our model performs. While this much monitoring may seem excessive, it is the best way to verify our model isn't overfitting. Overfitting happens when the model is performing really well on the training dataset, but isn't able to score well when tested. This signifies it isn't able to correcly generalize to new data and is basically useless in most scenarios. Generally overfitting happens when we have very large models parameter-wise, but the function we have to approximate isn't very complex, or the number of samples in the dataset is very small. We can see the difference in the loss curves between our model and a model which has way more layers and neurons in figure 3.4.

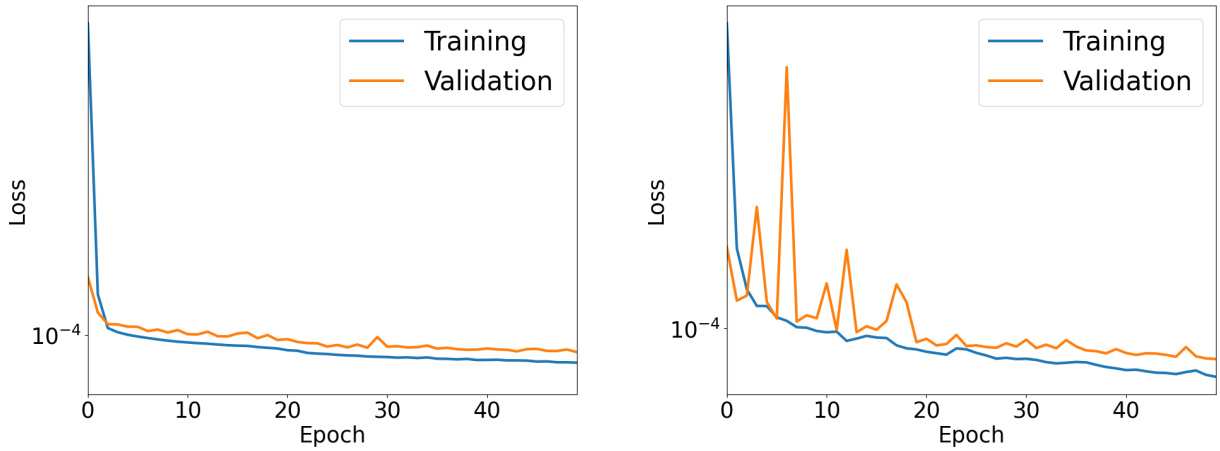

Figure 3.4: Plot of our model loss on the left, compared to a much bigger model on the right. The difference between the training loss and the validation loss gives us a visual representation of how much the model is overfitting. In particular we can see that the bigger model tends to overfit a lot more than our smaller model.

Once the training is complete, we can extract the values taken by each of our signals in the latent layer neurons and visualize them, as in figure 3.5. By comparing each feature with both energy (figure 3.7) and PSD values (figure 3.6) we can deduce what each of them represents. We can say with some confidence that feature 2 is directly anti-proportional to the energy of the signals and, while feature 3 has a similar proportionality, it probably represents Q_{long} as it slighly differs from the 2.2 plot when plotted against PSD. We can also see that feature 1 is the one best separating our data, as there seems to be almost a clear threshold around 0 where from one group it switches to the other. We can apply the cut method used traditionally with the new features, and in particular we decided to apply it to feature 1 and 2. We can see in figure 3.8 how this cut results in a much different discrimination in the overlapping area than figure 2.3.

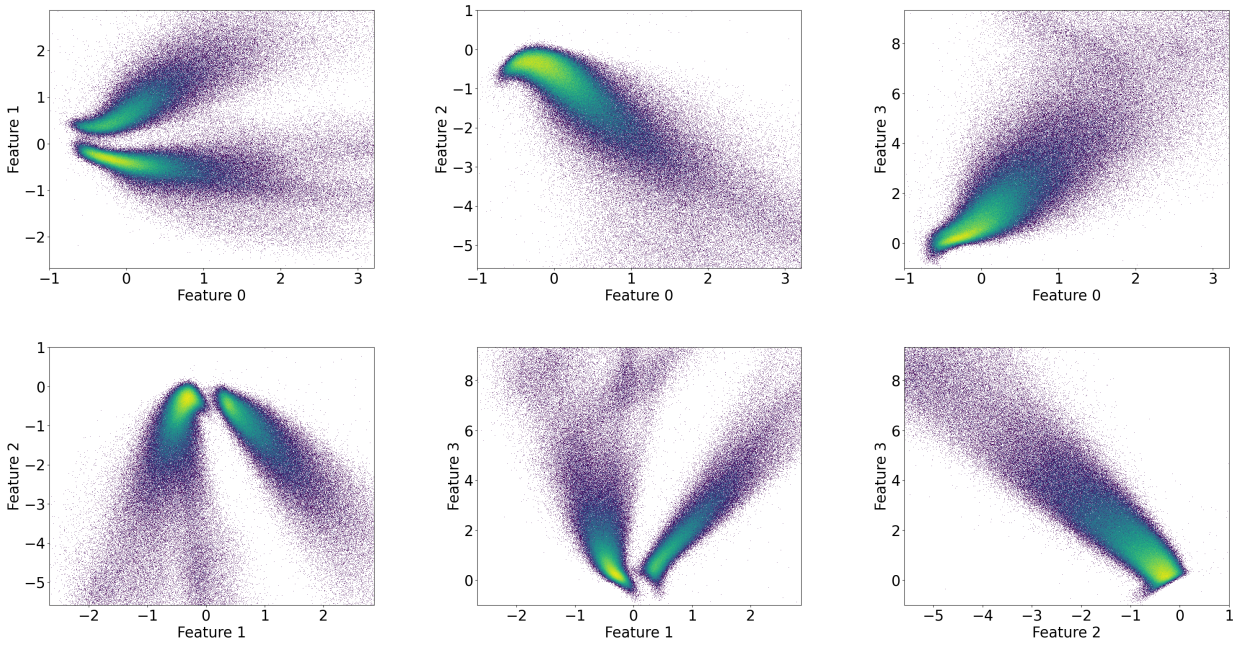

Figure 3.5: All the features plotted against each other

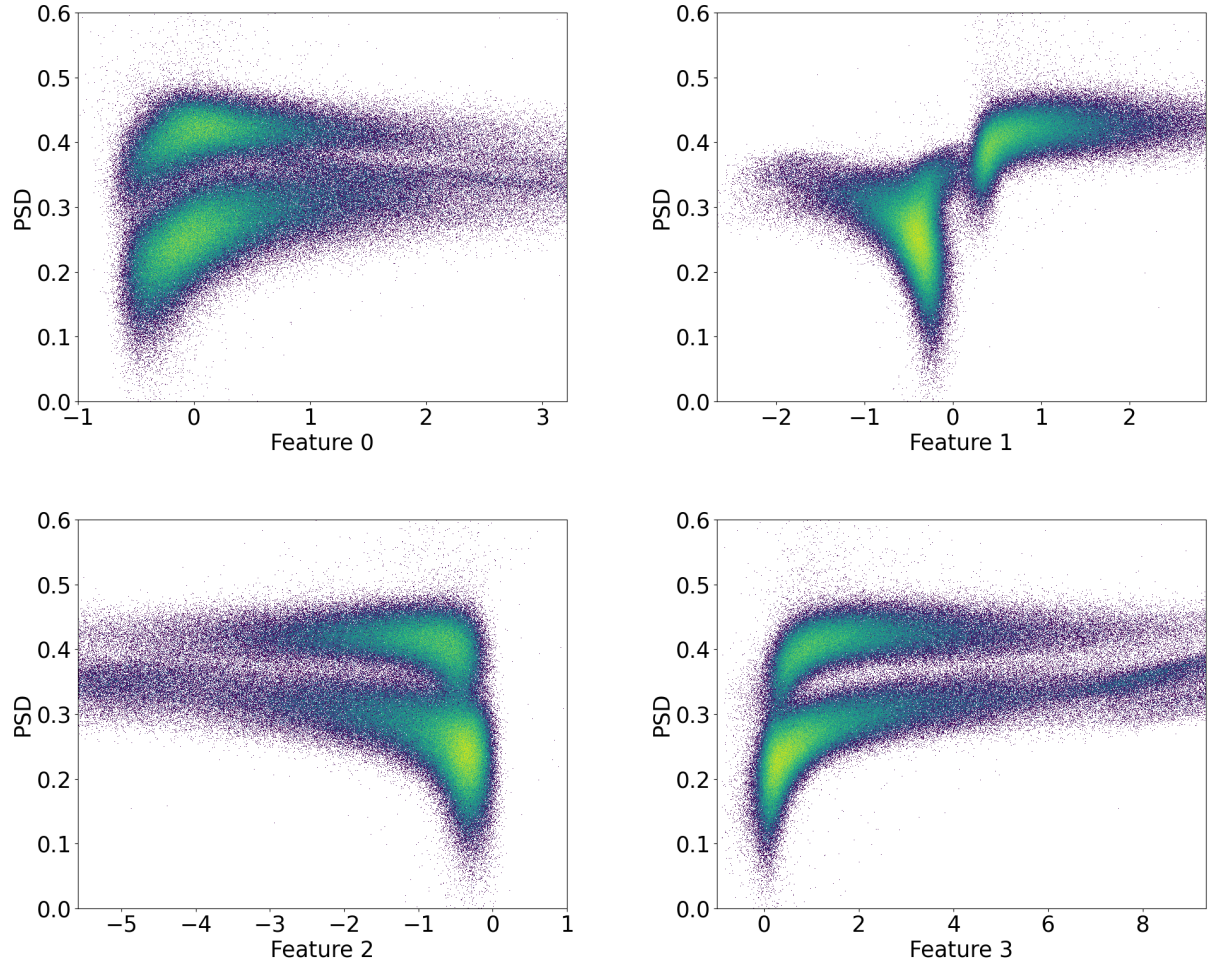

Figure 3.6: All the features plotted against PSD

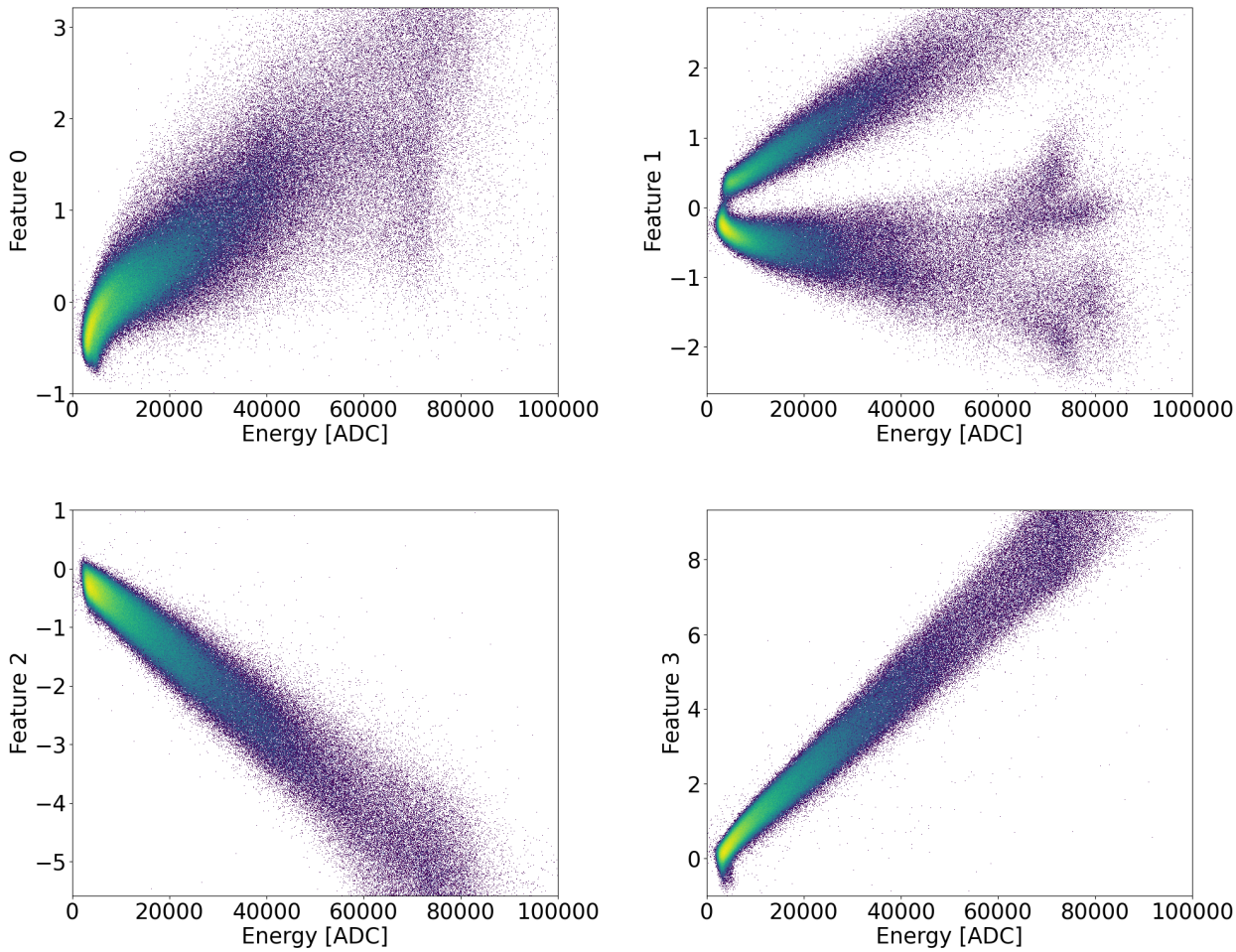

Figure 3.7: All the features plotted against energy

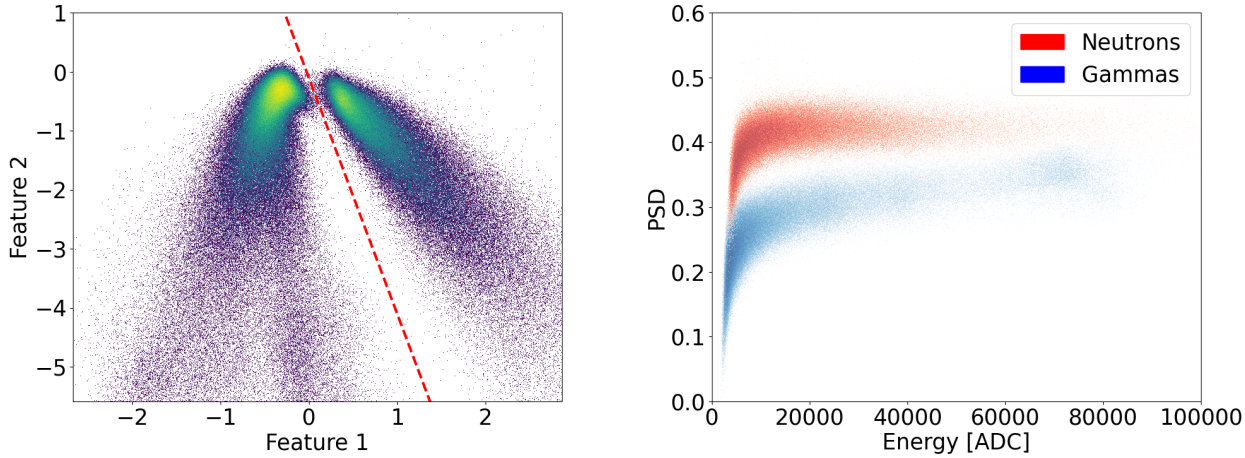

Figure 3.8: Plot of the cut we used on the left and the resulting classification on the right

	# of neutrons	%	# of gammas	%	Total
Old cut	348346	36.1	615776	63.9	964122
New Cut	369237	38.3	594885	61.7	964122

Table 3.2: Comparison between the two cut methods

3.2 Applying clustering algorithms

While doing a cut is a pretty simple but effective method, we are still introducing some biases in our analysis with it. To avoid them we could apply another type of unsupervised learning algorithm: clustering algorithms. Their main purpose is to classify each point in a dataset based on a certain similarity measure.

3.2.1 K-means

The first algorithm we used is K-means. It consists of generating K cluster means $\{\boldsymbol{\mu}_k\}_{k=1}^{K}$ and assigning each point $\boldsymbol{x}_n \in X$, where X is our dataset with cardinality N, to the one they are closest to based on the Euclidean distance. Its training is based upon the minimization of the following cost function:

$$\mathcal{C}(X, \{\boldsymbol{\mu}\}) = \sum_{k=1}^{K} \sum_{n=1}^{N} r_{nk} (\boldsymbol{x_n} - \boldsymbol{\mu}_k)^2 \tag{3.3}$$

where $r_{nk} = 1$ if x_n is assigned to the cluster k, $r_{nk} = 0$ otherwise. The whole training consists of alternating the update of the means $\boldsymbol{\mu}_k = \frac{1}{N} \sum_{n=1}^{N} r_{nk} \boldsymbol{x}_n$, and the update of the assignments r_{nk} [9]. We first apply this algorithm to each of the possible combinations between the extracted features, the energy and the PSD. We can see in figure 3.9 how this method fails to correctly identify our groups in most of the combinations. We then tried to apply it to all the features of our data (energy and PSD included) but it showed no real improvement, if not straight up got worse: as we can see from figure 3.10, we lost the only useful classification we had in the plot comparing feature 0 and 1. This is probably due to the simplicity of the algorithm and the fact that our data is pretty much uniform distance-wise but varies a lot density-wise.

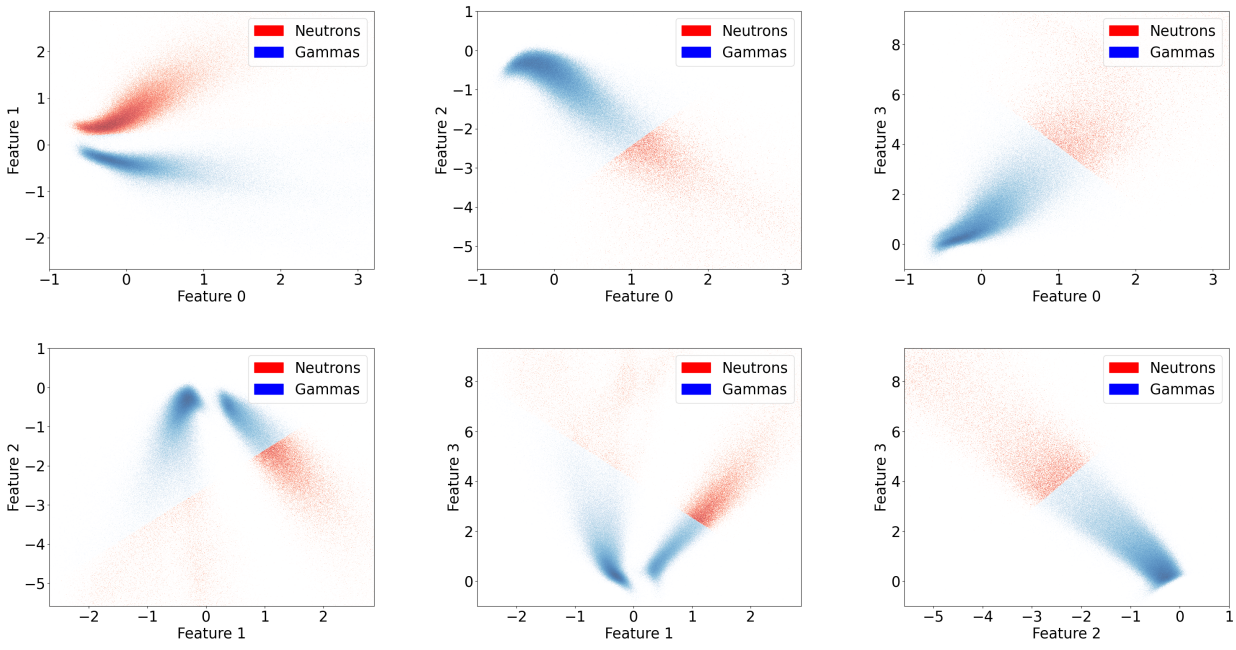

Figure 3.9: K-means clustering for combination of features

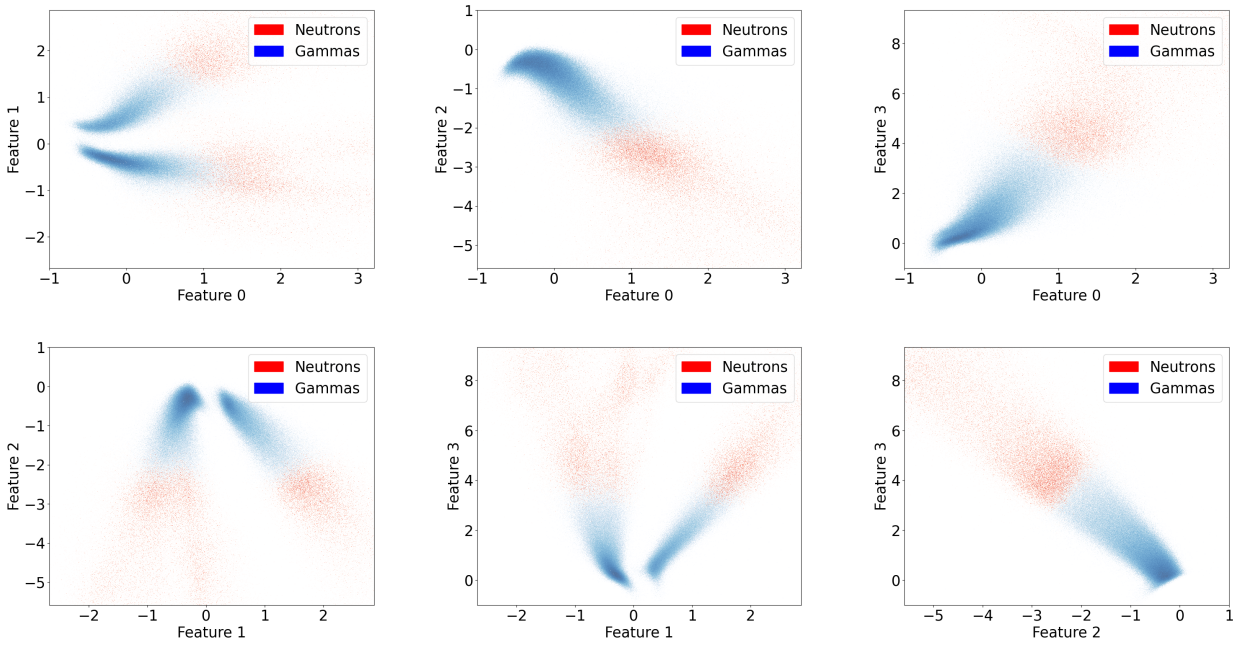

Figure 3.10: K-means clustering for the whole feature space

3.2.2 Gaussian Mixture Model clustering

We then used the GMM (Gaussian Mixture Model) clustering algorithm. It consists of fitting a mixture of Gaussian distributions to our dataset and each distribution represents a different label. In a D-dimentional space, a model with K distributions is described by the following density function:

$$p(\boldsymbol{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \quad \text{with} \quad \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \frac{1}{(2\pi)^D |\boldsymbol{\Sigma}_k|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_k)\right) \tag{3.4}$$

where $\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ is the k-th Gaussian density function with mean $\boldsymbol{\mu}_k$ and covariance matrix $\boldsymbol{\Sigma}_k$. π_k is the weight coefficient of the k-th distribution ($0 \leq \pi_k \leq 1$, $\sum_{k=1}^{K} \pi_k = 1$). To ease the notation we will group all the parameters into $\Theta_K = \{k \in [1, K] | \pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}$. The optimal parameters are computed via the expectation-maximization (EM) algorithm that consists of alternatively assign all points in the dataset the probability to be in the k-th cluster, and update the parameters based on those assignments. This process is repeated until the log-likelihood $\mathcal{L}(\Theta_K) = \sum_{i=1}^{N} \log(p(\boldsymbol{x}_i|\Theta_K))$ growth slows down under a certain ϵ per iteration [8]. As for the K-means algorithm, we applied GMM to the same combinations and then to the whole feature space. In figure 3.11 we can observe how the clustering was successful just for features 0 and 1, as it is the plot with the most Gaussian-like distribution of data, but fails to correctly identify the two groups in all other combinations. However, when GMM is applied to the whole feature space, its results get arguably worse as it isn't able to find any meaningful cluster. This is proably due to the fact that in 6 dimentions, the little resemblance to a Gaussian distribution that some plots had is completely lost.

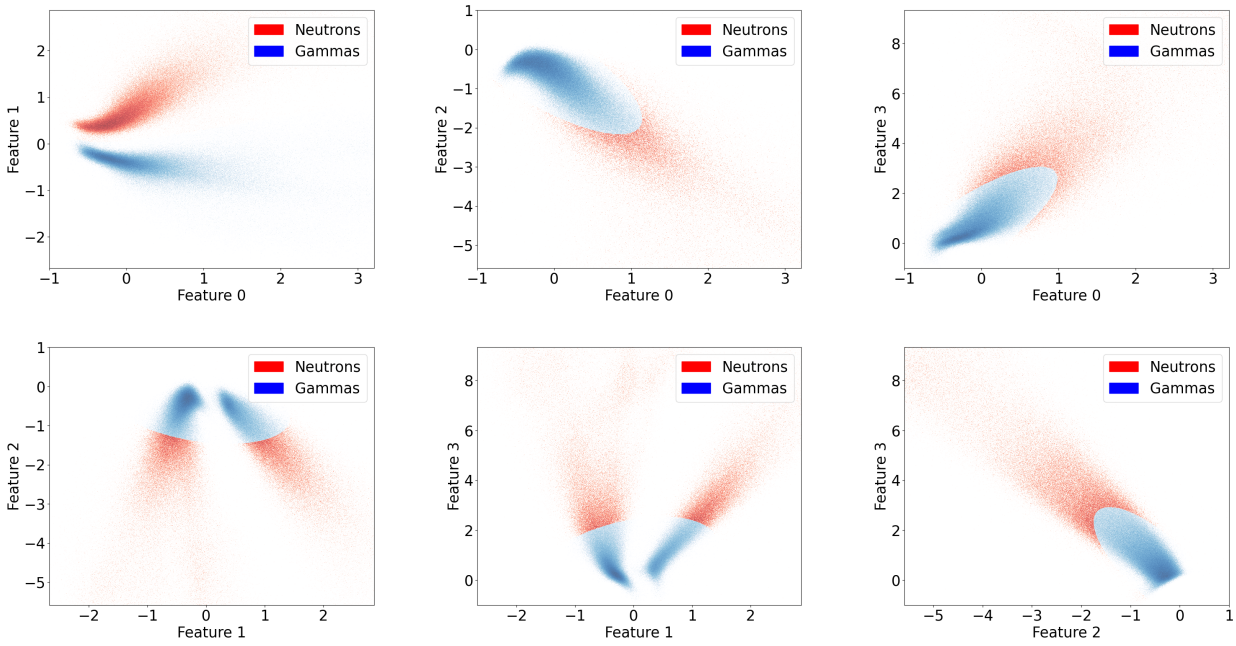

Figure 3.11: GMM clustering for combination of features

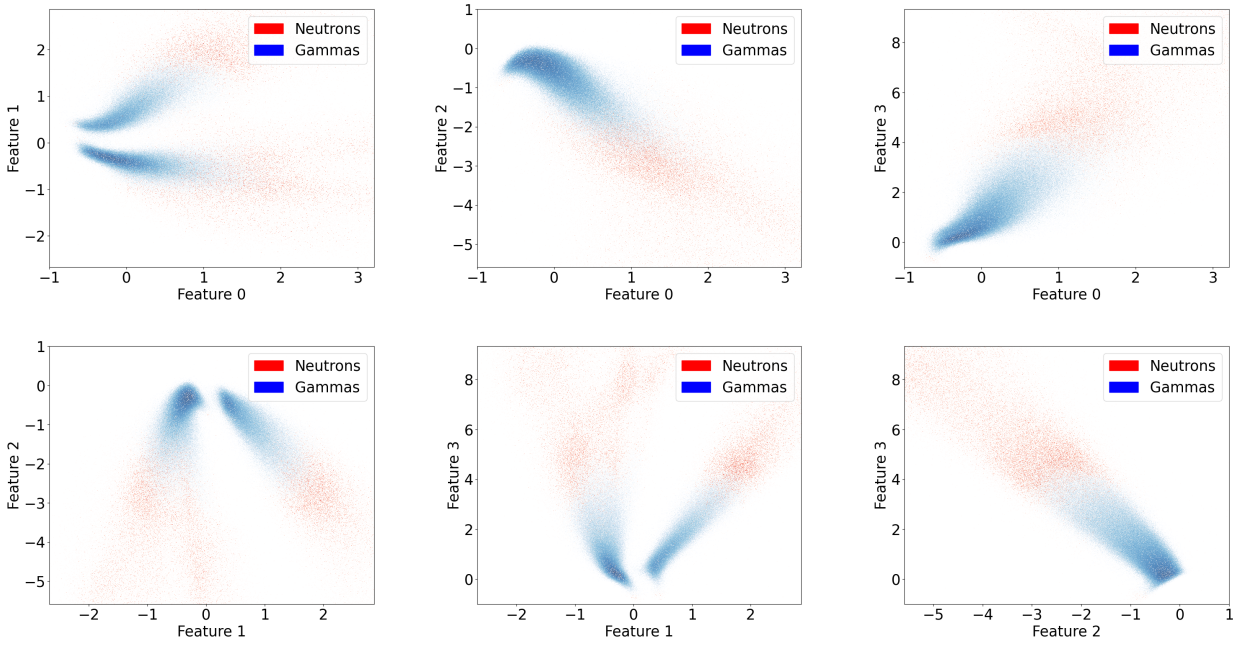

Figure 3.12: GMM clustering for the whole feature space

3.2.3 DBSCAN

In the end we tried the DBSCAN (Density-based spatial clustering of applications with noise) algorithm. It is initialized by choosing a distance metric $d(\cdot, \cdot)$, which we considered as the Euclidean one, and two other parameters: ϵ and minPts. We say that a point $\boldsymbol{x}_n$ is a *core-point* if there are at least minPts points inside it's ϵ-neighborhood $N_\epsilon(\boldsymbol{x}_n) = \{\boldsymbol{x} \in X | d(\boldsymbol{x}, \boldsymbol{x}_n) < \epsilon\}$. If a point $\boldsymbol{x}_i$ is inside the ϵ-neighborhood of another point $\boldsymbol{x}_n$, we say that $\boldsymbol{x}_i$ is *density-reachable* from $\boldsymbol{x}_n$ and they are part of the same cluster. Each cluster has at least one *core-point*. All the points that are not part of a cluster are labelled as outliers or noise [9]. As the other two algorithms, we tried it first on the combinations and then on the whole feature space. As the number of clusters isn't manually specified, it is usual to get tens or even hundreds of clusters in a single plot. To improve visualization and

labelling, we decided to apply GMM clustering to the points that where labelled as part of a cluster. As we can see in figure 3.13, for the combinations, DBSCAN generally performed better than the other two methods, while still having problems correcly identifying the clusters when working on plots where the distinction between the two groups isn't obvious. On the other hand, when we consider the whole feature space, it is clear from figure 3.14 that it is the best performing model. We can now see what these clusters represent in the PSD and energy space in figure 3.15. In table 3.3 we compare these results with what we got from applying the cut method on the features, considering signals with energies lower than $E_0 = 8000$ ADC.

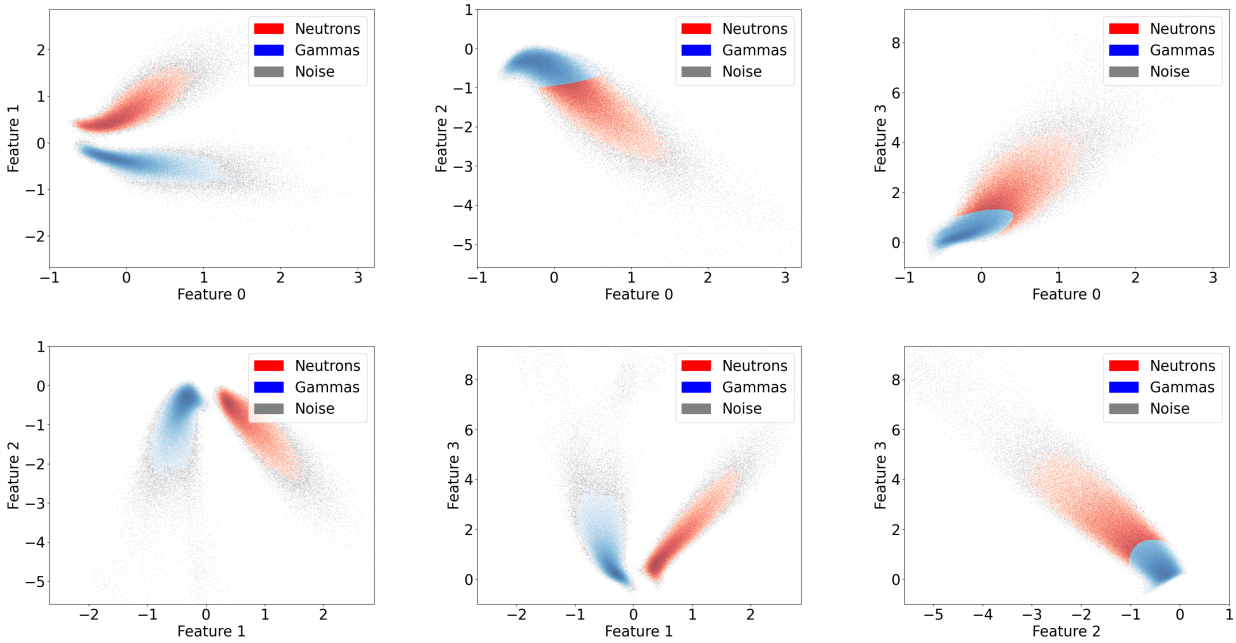

Figure 3.13: DBSCAN clustering for combination of features

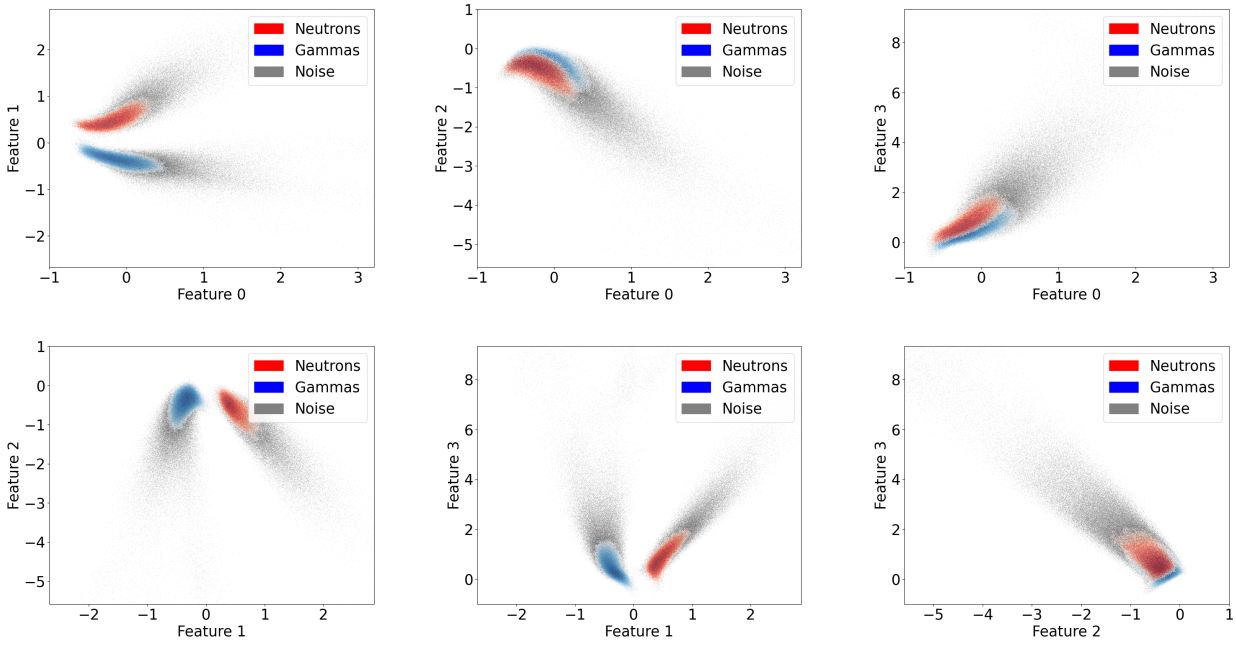

Figure 3.14: DBSCAN clustering for the whole feature space

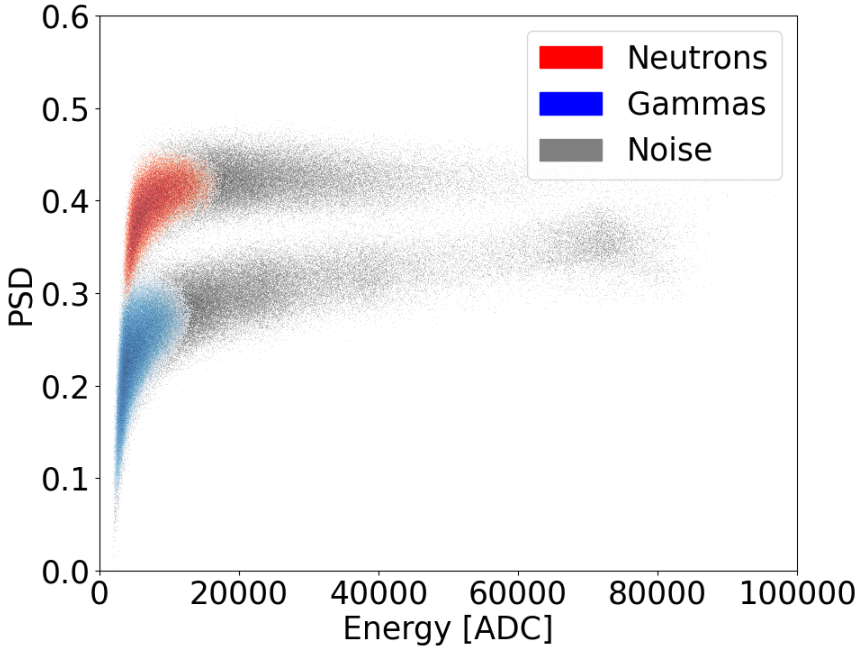

Figure 3.15: Representation of DBSCAN clustering for the whole feature space in the PSD vs energy plot

	# of neutrons	%	# of gammas	%	# of unlabelled	%	Total
Cut	85398	23.6	276032	76.4	0	0	361430
DBSCAN	64353	17.8	234893	65.0	62184	17.2	361430

Table 3.3: Comparison between the cut method and the DBSCAN method for $E < 8000$ ADC

3.3 Suppor Vector Machine

While the cut method is able to classify eventual new data, we can not extrapolate anything from the clustering methods in this regard. To perform predictions based on the DBSCAN classification we have to implement another machine learning algorithm: the Suppor Vector Machine. A SVM is a supervised machine learning algorithm with the purpose of finding the function $f(\boldsymbol{x})$ that best separates two classes in a dataset. In its simplest implementation this function is basically an hyperplane in the feature space:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \cdot \boldsymbol{x} + \boldsymbol{b} \tag{3.5}$$

where $\boldsymbol{b}$ is the bias term and $\boldsymbol{w}$ are the weights. This implementation works only with linearly separable data, that's why it is common to "upgrade" a SVM to a kernel machine using the so called "kernel trick", which consists of projecting the input space $\boldsymbol{x}$ into a different space $\phi(\boldsymbol{x})$ where the data becomes linearly separable [4]. For our purposes, if we consider the whole feature space, we can use the basic SVM implementation and still obtain great results. We must note that this model only works when the signals we want to predict the nature of have energies in the clustered range. It could be used to classify signals in the higher energy ranges but we must note that its accuracy deteriorates greatly. We can now compare the SVM and the cut methods and, as we can see from table 3.4, they differ just by a few hundred samples in their classifications, while being applied to hundreds of thousands of them. This probably suggests that, even if the cut method may conceal some biases, it is still a reasonable method to discriminate between gamma rays and neutrons signals.

	# of neutrons	%	# of gammas	%	Total
Cut	85398	23.63	276032	76.37	361430
SVM	85527	23.66	275903	76.34	361430

Table 3.4: Comparison between the cut method and the SVM for $E < 8000$ ADC

Chapter 4

Conclusions

In this work we tried to improve the neutron/gamma discrimination of scintillator signals for the study of reactions such as the $^{22}\mathrm{Ne}(\alpha, n)\,^{25}\mathrm{Mg}$ one.

After showing the limitations of traditional Pulse Shape Discrimination tecniques, incapable of correct low energy classification, we trained an autoencoder neural network and extracted the latent layer values, giving us more parameters to base our discrimination upon. The first classification method we used is a simple cut made in the feature space. To avoid eventual biases concealed in the cut method, we performed clusterization via three different algorithms. The K-means and the Gaussian Mixture Model clustering algorithms proved to be incapable of correcly identifying the neutron and gamma groups. The DBSCAN algorithm however was able to clusterize properly the signals in the lower energy ranges. To perform predictions based on this classfication we then trained a Support Vector Machine and compared it to the cut method used before.

Bibliography

[1] Chemseddine Ananna et al. "Intrinsic background of EJ-309 liquid scintillator detectors". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 1060 (2024), p. 169036. ISSN: 0168-9002. DOI: https://doi.org/10.1016/j.nima.2023.169036. URL: https://www.sciencedirect.com/science/article/pii/S0168900223010367.

[2] G. F. Ciani et al. "Direct Measurement of the $^{13}C(\alpha, n)^{16}O$ Cross Section into the s-Process Gamow Peak". In: *Phys. Rev. Lett.* 127 (15 Oct. 2021), p. 152701. DOI: 10.1103/PhysRevLett.127.152701. URL: https://link.aps.org/doi/10.1103/PhysRevLett.127.152701.

[3] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. 2016. arXiv: 1511.07289 [cs.LG]. URL: https://arxiv.org/abs/1511.07289.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[5] J Griffiths et al. "Pulse shape discrimination and exploration of scintillation signals using convolutional neural networks". In: *Machine Learning: Science and Technology* 1.4 (Oct. 2020), p. 045022. DOI: 10.1088/2632-2153/abb781. URL: https://dx.doi.org/10.1088/2632-2153/abb781.

[6] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[7] Glenn F. Knoll. "Radiation detection and measurement". In: *Proceedings of the IEEE* 69 (1981), pp. 495–495. URL: https://api.semanticscholar.org/CorpusID:16661089.

[8] Genghui Li et al. "Offline and Online Objective Reduction via Gaussian Mixture Model Clustering". In: *IEEE Transactions on Evolutionary Computation* 27.2 (2023), pp. 341–354. DOI: 10.1109/TEVC.2022.3168836.

[9] Pankaj Mehta et al. "A high-bias, low-variance introduction to Machine Learning for physicists". In: *Physics Reports* 810 (May 2019), pp. 1–124. ISSN: 0370-1573. DOI: 10.1016/j.physrep.2019.03.001. URL: http://dx.doi.org/10.1016/j.physrep.2019.03.001.