



**ALGORITMO DE RECONOCIMIENTO DE
PATRONES DE SEÑALES DE MATERIA
OSCURA EN PÍXELES DE LOS
DETECTORES DE DAMIC-M**
(Pattern recognition algorithm for Dark
Matter signals on pixel detectors DAMIC-M)

Trabajo de Fin de Máster
para acceder al

MÁSTER EN CIENCIA DE DATOS

Autora: Silvia Magdalena López Monzó

Director\es: Ignacio Heredia Cacha y Nuria Castello-Mor

Junio - 2020

Contents

1	Abstract	2
2	Introduction to Dark Matter and DAMIC-M	2
3	Artificial neural networks	3
3.1	Convolutional neural networks	8
3.2	Metrics	11
4	Data preprocessing and tasks	13
4.1	Original data.....	13
4.2	Data preprocessing	15
4.3	Classification and localization tasks	17
5	Model details	18
5.1	Models to classify	18
5.2	Models to classify and locate.....	19
6	Training details	21
6.1	Models to classify	21
6.2	Models to classify and locate.....	22
7	Results	23
7.1	Analysis of results without noise added	23
7.1.1	Model to classify into electrons, muons, alphas and noise.....	23
7.1.2	Model to classify into electrons, muons and noise	24
7.1.3	Model to classify into electrons and muons	24
7.1.4	Model to classify and locate electrons, muons and alphas.....	29
7.2	Analysis of signals with noise	30
7.2.1	Model to classify into electrons, muons, alphas and noise.....	30
7.2.2	Model to classify into electrons, muons and noise	31
8	Conclusions	32
9	Further work	33
	References	33

1 Abstract

Scientific observations lead to postulate the existence of dark matter and great efforts are being carried out by the scientific community to prove its existence. For this aim, the DAMIC-M (Dark Matter In CCDs) Experiment was built. In this thesis, Deep Learning is applied to achieve particle identification, in particular, convolutional neural networks will be built. The main goal of these models is to determine whether a particle of the Standard Model has been detected or not, hence to discard this event as a dark matter signal.

2 Introduction to Dark Matter and DAMIC-M

The study of the fundamental constituents of matter is the subject of particle physics. Our understanding of this branch of physics is based on the description provided by the Standard Model. The model describes how the universe is made up of a few basic blocks, these are the fundamental particles governed by four forces: gravitational, electromagnetic, strong, and weak. The Standard Model has been tested for several decades to conclude that the majority of the observed processes can be described by it, thus showing its potential [1]. Unluckily, a number of phenomena observed in nature is lacking an explanation, for instance, dark matter.

Astronomical and cosmological observations, such as the rotation curves of galaxies and the cosmic microwave background lead us to postulate the existence of dark matter, beyond the Standard Model. This discovery would be a huge breakthrough for the scientific community as we only know how to describe the properties and interactions of ordinary matter which constitutes 5% of the total content of the universe [2].

Although there are many well-motivated theories providing candidates for dark matter, we will focus on the most studied candidate in the literature: Weakly Interacting Massive Particles (WIMPs). Around the world, many experiments are developed in order to discover these theorized particles, but no technique has been successful to prove their existence so far.

Our interest will be focused on the DAMIC-M Experiment: Dark Matter in CCDs, where CCDs stands for charge-coupled devices. The experiment employs the bulk silicon scientific-grade CCDs as a target for interactions of dark matter particles. The principle of detection with a CCD is illustrated in Figure 1. The charge produced due to the interaction between the ionizing particle and the silicon bulk, drifts towards the pixel gates, where it is held in place until the readout. After a given exposure time, the readout process starts and the charge is transferred vertically from pixel to pixel along each column until it reaches the last row, the so-called serial register. The signal is based essentially on i) ionization signals produced by the interaction of Standard Model particles with the silicon bulk of the CCDs, ii) the intrinsic detector noise composed of the dark current (from thermal excitation from charge released by traps in the surface and bulk of the detector material, or produced by ionizing backgrounds

particles) and electronic noise (during readout) and iii) (if we are lucky) the dark matter signals, through absorption or nuclear/electronic recoil [3]. In other words, these signals from ionizing particles, known as tracks, are energy deposited in silicon pixels along the particle trajectory within a CCD.

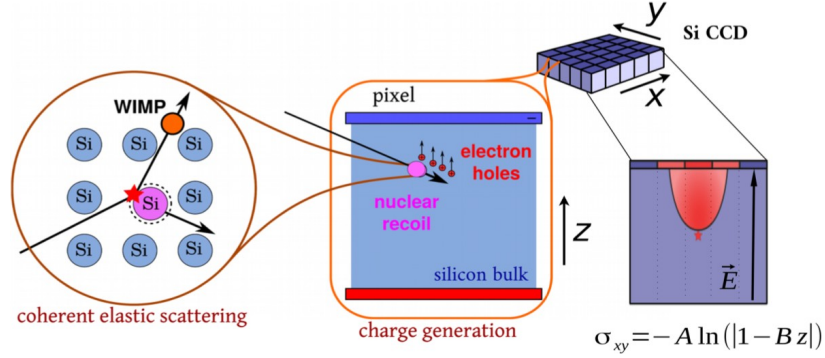


Figure 1: Internal process of a CCD when an ionizing particle interacts with its silicon bulk [3]

By virtue of the low readout noise of the skipper CCDs, in conjunction with low dark current, will allow DAMIC-M to observe physics process with collisions energies as low as 1eV. Most of the dark matter interactions with the silicon bulk result in the production of few charges [4], where a **dark matter signal can be seeing as a distortion of the intrinsic detector noise**. It is then extremely important to properly understand the expected background from Standard Model particle interactions to be able to discriminate it from the dark matter signal coupled with the intrinsic detector noise.

For this reason, the goal of the thesis is to develop a pattern recognition algorithm based on Deep Learning techniques to provide Standard Model particle identification, in order to help to discriminate the Standard Model noise from the coupled dark matter and intrinsic signal events. The major challenge for the algorithm will be to prove the viability of disentangling the nuclear and/or electronic recoil due to dark matter interaction with the silicon bulk of the intrinsic detector's noise.

3 Artificial neural networks

Parallel computation and neural networks are the new computing paradigms in machine learning. Machine learning algorithms are trained rather than explicitly programmed, as they can find statistical structure in input data that eventually allows the system to come up with rules for automating the task [5]. The key element of neural networks is their structure: a large number of interconnected processing elements, so-called **neurons**, working in parallel enabling multitasking [6].

Artificial neural networks are models inspired by biology. They can represent knowledge at an abstract level based on the architecture of the network and the connections between the neurons.

Neurons are organized in **layers**, starting with the input layer which will acquire the data and ending with the output layer to return the results. If hidden layers are added between the input and output layers, the neural network will be called **deep neural network** and the depth of the model will be given by the number of hidden layers.

Deep neural networks are described by **deep learning**, which is a specific subfield of machine learning. It stands for the fact of learning many successive layers of representations, whereas other approaches to machine learning tend to focus on learning with one or two layers. Its importance lies in the fact that it is possible to automate feature engineering. Other machine learning techniques involve transforming the input data into one or two successive representation spaces, so this implies a previous work of extracting interesting features of the data in order to contribute to the classification. Deep learning learns all features in one step that is automated.

It must be mentioned that a neural network can have different topologies such as multilayer networks, competitive networks and recurrent networks. Before we dig any further into these topologies, we need to understand how a single neuron works.

In Figure 2, we can observe the **process of a single neuron** with a given nonlinear activation function, $f(x)$. The origin of this function comes from biology once again, so when the stimulus that a neuron receives exceeds a certain threshold, ϑ , then the neuron emits an impulse; otherwise, it remains at rest. This idea is implemented in the artificial neural network by calculating the weighted sum of the inputs (x_j) received by the neuron, and then, by filtering it with a threshold-type activation function to obtain the output, y .

Mathematically, the neuron performs a simple computation given the inputs to obtain the output value as follows:

$$y = f\left(\sum_{j=0}^n w_j x_j\right) \xrightarrow{\text{for } i \text{ neurons}} y_i = f\left(\sum_{j=1}^n w_{ij} x_j - \vartheta_i\right) = f\left(\sum_{j=0}^n w_{ij} x_j\right) \quad (1)$$

where w_{ij} stands for the weights (can take both positive and negative values) of the neuron i for the each input x_j ; $f(x)$ is the, already mentioned, *activation function* and ϑ_i is the activation threshold.

The activation threshold value ϑ_i can be included in the sum considering a new auxiliary neuron denoted by $x_0 = -1$ connected to y_i with $w_{i0} = \vartheta_i$ [7].

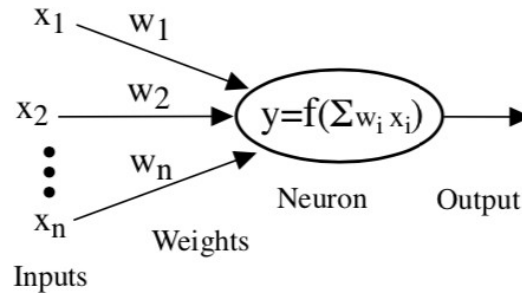


Figure 2: Internal process of a single neuron to produce the output as a result of applying an activation function to a linear combination of weights and inputs [6]

The most popular continuous **activation functions** are described in Table 1:

Linear functions	$r(x) = x$
Step functions	$\text{sgn}(x) = \begin{cases} -1 & x < 0 \\ +1 & \text{otherwise} \end{cases} \quad \text{step}(x) = \begin{cases} 0 & x < 0 \\ 1 & \text{otherwise} \end{cases}$
Sigmoidal functions	<p>Logistic function $f_c(x) = \frac{1}{1+\exp^{-x}} \in [0, 1]$</p> <p>Hyperbolic tangent function $f_c(x) = \tanh(x) \in [-1, 1]$</p>
Rectified linear unit (ReLU)	$f(x) = \max(0, x)$

Table 1: Activation functions for neural networks

Step functions give a binary output depending only on the position for a given threshold value and sigmoidal functions are bounded monotonic functions which return a nonlinear output for the inputs.

Once the different activation functions have been discussed, in Figure 3, we will explain some of the possible **topologies** of neural networks.

The topology on the left in Figure 3 is the *multilayer network*, which is formed by an input layer, a number of hidden layers, and an output layer. Each of the hidden and output neurons receives an input from the neurons on the previous layer, so this topology shows only connections between neurons in consecutive layers.

The network in the middle shows a *recursive architecture*, that unlike the multilayer network, allows previous outputs to be used as inputs.

Finally, in the *competitive architecture*, the layers are connected with consecutive layers but there are connections between neurons in the last layer. In this last layer, only one of the neurons will be activated (winner takes all).

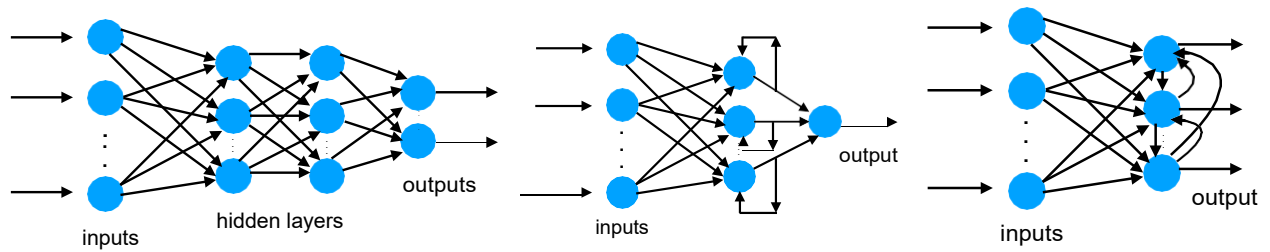


Figure 3: From left to right, different topologies for a neural network: multilayer, recursive and competitive

It is important to highlight that the number of hidden layers will define whether the architecture of the network is deep or not. Therefore the topologies described can be used in Deep Learning if there is at least, one hidden layer.

Topologies and activation functions define the neural network, and the weights among the neurons are the parameters that the network has to learn by fitting them to the input data. During this learning process, the network will be able to generalize the knowledge learned.

The different **learning processes** can be classified into four broad categories: supervised, unsupervised, self-supervised and reinforcement learning. We shall delve into the first two:

- **Supervised learning:** input data $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_n)$ is trained with its correspondent labels $\mathbf{Y} = (y_1, \dots, y_n)$. These labels classify each data point into one or more groups. The networks learn the distribution of the data and trains itself. The weights are calculated by minimizing an error function (optimization problem), which computes the difference between the desired output values and those predicted by the network. An error that can arise in this type of learning is the error convergence, as the error function may present multiple local minima where the network can get stuck and hence, not reach the optimal global minimum. Supervised learning is used for classification and regression problems.
- **Unsupervised learning:** input data $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_n)$ is trained without any labels, therefore, the network must discover by itself patterns or categories. This type of network is used for clustering problems, where each neuron belonging to the last layer represents a cluster.

Once the neural network is computed, it is important to check the quality of the resulting model. As discussed before, in the case of supervised learning, a measure of the quality can be given by the errors between the desired and the computed output values for the training data.

The resulting value is the so-called **loss score** and the measure most commonly used is the Root Mean Square Error (RMSE) defined as:

$$rmse = \sqrt{\frac{1}{n} \sum_{i=1}^n \|y_i - \hat{y}_i\|^2} \quad (2)$$

where \hat{y}_i is the network output for the input vector x_i , whereas y_i represents the true label.

For the purpose of the RMSE's minimization, several algorithms were developed. The most famous iterative algorithm is the so-called **stochastic gradient descent**, which is used as the **optimizer** of the network. During each epoch (each iteration over all the training data), the weights are changed with the goal of minimizing the error. Therefore, at each iteration step, the weights are modified proportionally to the negative gradient of the error function. Note that in the first epoch, weights are assigned with random values. Computing the gradient of the loss with regard to the network's parameters is called the *backward pass*.

In order to verify the generalizability of the model, **cross-validation** is needed. To this aim, the splitting of the available data into three sets is required: one part for training, another for validating and the last one for the final test. Cross-validation is an iterative process where the model is evaluated on different partitions, and yields the arithmetic mean of all the evaluations performed. These partitions are generated from the training and validation data, where the training of the model is performed on the training dataset and its evaluation on the validation dataset.

When the test error is much larger than the training error during the training process, we observe **overfitting**. When this problem appears, the model is not able to capture the real trends of the population¹, even though it may present a small training error. In this case, the model is not able to generalize well when applied to the test data. Overfitting is a critical problem in neural networks, in order to solve it, the network should be carefully designed and/or regularization techniques should be adopted. One of the most used techniques is the **dropout** applied to some layers. Dropout consists of randomly setting to zero a number of output features of the layer during training. In each iteration of the training, different neurons will get affected by the dropout. This regularization technique simplifies the network and avoids some neurons to over-specialize, avoiding overfitting. The only parameter of this layer is the *dropout rate*, which is the fraction of neurons that will have zero as output during each iteration of the training.

¹The model captures the trends of the sample which consists of one or more observations drawn from the population

Another solution to fix overfitting is the **early stopping**, which consists of interrupting training when the validation loss is no longer improving, thus avoiding having to train for more epochs than necessary.

In order to understand how a neural network works, Figure 4 displays the internal **workflow of a deep neural network**. First of all, input data is injected into the network through the first layer where it will experience transformations according to the random weights associated with the layer. These data transformations will be learned by the network in order to find the proper weights for each layer. To do so, and to control the predictions, the network will compute a loss score using a chosen loss function, the predictions and the true targets. Note that this score is used as feedback to adjust the value of the weights and this process is done by the optimizer. Then, the weights will be updated and the training loop will start again.

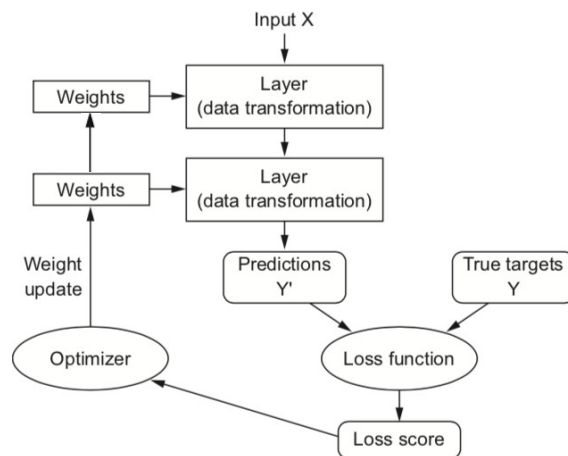


Figure 4: Internal process of deep neural network with two hidden layers [5]

This same internal process occurs also inside a classical (no deep) neural network. In this case, no hidden layers would be used and the weights would be located between the input X and the predictions Y^0 .

Deep learning has been described so far, hence a specific type of deep neural network will be discussed: convolutional neural networks. We will focus our interest on these as they will be used for the aim of this thesis.

3.1 Convolutional neural networks

In deep learning, a **convolutional neural network** (CNN or Convnet) is the model most commonly applied to computer vision applications.

The fundamental difference between a densely connected layer (classic neuronal layers discussed previously) and a convolution layer is that dense layers learn global patterns from their input feature space, whereas convolution layers learn local patterns [5]. Once this difference is clear, we can discuss the main characteristics of convnets:

- Patterns learned are **translation invariant**: when a pattern is learned in a certain location and then appears again in another different location, there is no need to learn the already known pattern again. Note that a dense network should learn it once again.
- Ability to learn **spacial hierarchies**: earlier convolutional layers encode basic features, whereas layers higher up encode complex features by combining the previous basic features.

Convolutional layers are composed of **filters** with two spatial axes (height and width) and a third dimension representing the number of channels. These layers operate over 3D tensors, (height, width, number of channels). The number of channels depends on the nature of the image: if the input data is an RGB image, the dimension of the depth axis is 3, because the image has three color channels: red, green, and blue. Otherwise, if it is a black and white picture, the dimension is 1 due to the levels of gray. Accordingly, the number of channels of the filters is given by the input they receive.

Therefore, each convolutional layer is defined by two parameters: the number of filters applied and the shape of these filters.

Once the components of the convolutional layer are presented, we shall continue with its inner functions.

As mentioned before, the number of filters and their dimensions are the key parameters of convolutions, since the number of filters defines the depth of the output feature map and the dimensions of them, define the size of the patches extracted from the inputs. Let us follow an example in Figure 5 to understand it better.

Given a black and white picture of shape (5×5), the input feature map will have as dimensions (5×5×1). For the first convolutional layer, it has been chosen one filter of dimensions (2×2). To create the output feature map, the convolution operation will slide the filter through all the possible locations in the input space. Finally, the dimensions of the output space in this example, will be (4×4×1). The result is still a 3D tensor, where the first two components are given by the dimensions of the filter and the last one, stands for the number of filters applied. If the input data was an RGB image, the filter would be applied to each of the three components and then perform a combination of these components to obtain one dimension. Therefore the third dimension no longer represents the colors, RGB or black and white, but the number of filters applied to the input space.

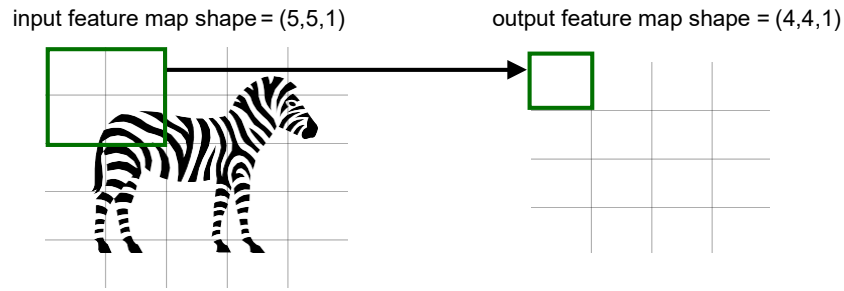


Figure 5: From the input space ($5 \rightarrow 5 \rightarrow 1$) to the output space ($4 \rightarrow 4 \rightarrow 1$) through one filter of shape ($5 \rightarrow 5$)

The main task of filters is to encode specific aspects of the input data, and for this reason, several filters are applied, instead of just one as shown in the previous example.

Figure 5 also introduces us to a problem of convolutional networks: border effects. The pixels located at the corners on the input feature map, would have less presence in the output whereas the pixels located in the center, would have more presence as they are included in several iterations of the sliding window. Another problem associated with the border effects is that the output feature map shrinks: the bigger the shape of the filter is, the smaller will be the output feature map. The solution is **padding** the input feature map. Padding consists of adding an appropriate number of rows and columns on each side of the input feature map so each pixel has equal weight in the output feature map.

Therefore, padding can influence output size, but also the concept of strides. The description of convolution assumed that the sliding of the windows is performed pixel by pixel. The distance between the windows is also a parameter of the convolution and is called **stride**, which defaults to 1. In Figure 6, on the left, we observe the default value and so, the windows are separated by one pixel. On the right, the strides take value 2, so the windows are now separated by two pixels. Note that the higher the strides, the smaller will be the shape of the output feature map. A live explanation of these operations can be found in [8].

The fact of modifying the strides and the padding creates changes on the dimensions of the output feature map². To downsample the output feature maps and hence, to reduce the number of parameters to process, instead of strides, max-pooling is usually used. **Max-pooling** extracts windows from the input feature maps and creates an output where each value will be the maximum value of each channel. The usual size of the windows used in the max-pooling layers is ($2 \rightarrow 2$).

²Note that the last dimension changes only depending on the number of filters applied

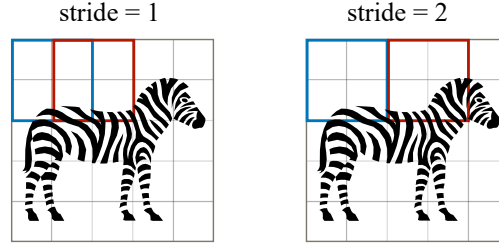


Figure 6: On the left, stride takes its default value; on the right, stride takes value two. Color blue means the first window taken and color red stands for the slid window in the second iteration

Max-pooling is not the only technique to downsample the output feature maps, it is possible to use average pooling instead, where each local input patch is transformed by taking the average value of each channel over the patch. Although, the performance of the max-pooling is better since the maximal presence of features orders more information than their average presence and max-pooling is also more efficient.

Once the features are extracted, this 3D tensor will be the input of a densely connected layer (or layers). But first, this vector must be transformed into a 1D vector through a **flatten** layer, which is necessary since classifiers only process vectors.

For the purpose of this thesis, convolutional neural networks will be used in order to classify particles on the DAMIC-M detector.

3.2 Metrics

Loss functions are used to compute a quantity that a model should seek to minimize during training. However, a loss score does not order much information, this is the reason why we use metrics to assess the performance of the model. Through this section, we will discuss the different metrics used.

For classification problems, we use accuracy, precision, recall, microaverage and macroaverage, which are defined as follows:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$recall = \frac{TP}{TP + FN} \quad (3)$$

$$precision = \frac{TP}{TP + FP}$$

where TP stands for True Positive, TN for True Negative, FP for False Positive and FN for False Negative. Figure 7 shows how these four values are distributed in a confusion matrix for a binary (two classes) classification problem, *Negative* and *Positive*.

		Predicted labels	
		N	P
True labels	N	TN	FP
	P	FN	TP

Figure 7: Confusion matrix showing the distribution of TN, FN, FP and TP for a binary classification problem

When we face a multiclass classification problem, averaging the evaluation measures can give a view on the general results. For this purpose, we introduce micro-averaged and macro-averaged results (equations 4).

$$\begin{aligned}
 macro-avg\ precision &= \frac{1}{n_c} \sum_{i=1}^{n_c} precision_i \\
 macro-avg\ recall &= \frac{1}{n_c} \sum_{i=1}^{n_c} recall_i \\
 &= \frac{\sum_{i=1}^{n_c} TP_i}{\sum_{i=1}^{n_c} (TP_i + FP_i)} \\
 micro-avg &= \frac{\sum_{i=1}^{n_c} TP_i}{\sum_{i=1}^{n_c} (TP_i + FP_i)}
 \end{aligned} \tag{4}$$

where n_c stands for the number of classes in the classification problem. The difference between both averages lies on the fact that macroaveraging gives equal weight to each class, whereas microaveraging gives equal weight to each classification decision [9]. As microaveraged results are more effective on the large classes in a test collection and our classification problems has maximum four classes, we would compute macroaveraged results.

For classification+location problems, we can use the metrics just mentioned for the classification task, but the location task requires the introduction of a new metric called Intersection over Union. Intersection over Union (IoU) is an evaluation metric used to measure the accuracy of an object detector on a particular dataset [10].

To calculate this metric, we may define some concepts:

- The ground-truth bounding boxes: the (x,y) coordinates, height and width of the bounding boxes which specify the location of the particle.
- The predicted bounding boxes from our model with the same four variables.

Therefore, IoU can be determined by dividing the area of overlap by the area of union (Figure 8). In the numerator, we compute the area of overlap between the predicted bounding box and the ground-truth bounding box. On the other hand, the denominator is the area of union of both predicted and ground-truth bounding boxes. The greater the overlap between the bounding boxes, the closer the IoU will be to the unit.


$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$


Figure 8: Definition of the Inverse over Union metric to evaluate a localization model [10]

4 Data preprocessing and tasks

4.1 Original data

Files containing the data are in .npz format which is a zipped archive of files named after the variables they contain in .npy format [11]. In this case, each .npz file represents an event containing two variables: energy of the particle and noise of the signal.

Both energy and noise data are numpy arrays composed of 300x300 pixels. Each pixel value represents the value of the energy measured in electronvolts (eV). Whereas the noise is generated from two independent distributions: i) Poisson distribution ($\lambda = 0.0003333$), also called dark current. ii) Gaussian distribution ($\mu = 0$, $\sigma^2 = 0.25$), also called electronic noise.

The file name gives information about the simulated event and the particle detected. For the aim of this thesis, we will be interested in the last digits as they represent the type of particle:

- xxxxxxxxxxxx pdg 11.npz: electrons
- xxxxxxxxxxxx pdg 1000020040.npz: alphas
- xxxxxxxxxxxx pdg 13.npz: muons

Each energy image contains the track of the particle. The first row of Figure 9 shows the track of an electron and the noise saved during such event. The next row displays the energy of a muon on the left, and an alpha decay event on the right. The noise stored has not been displayed as it looks similar to the noise saved during the electron signal already shown.

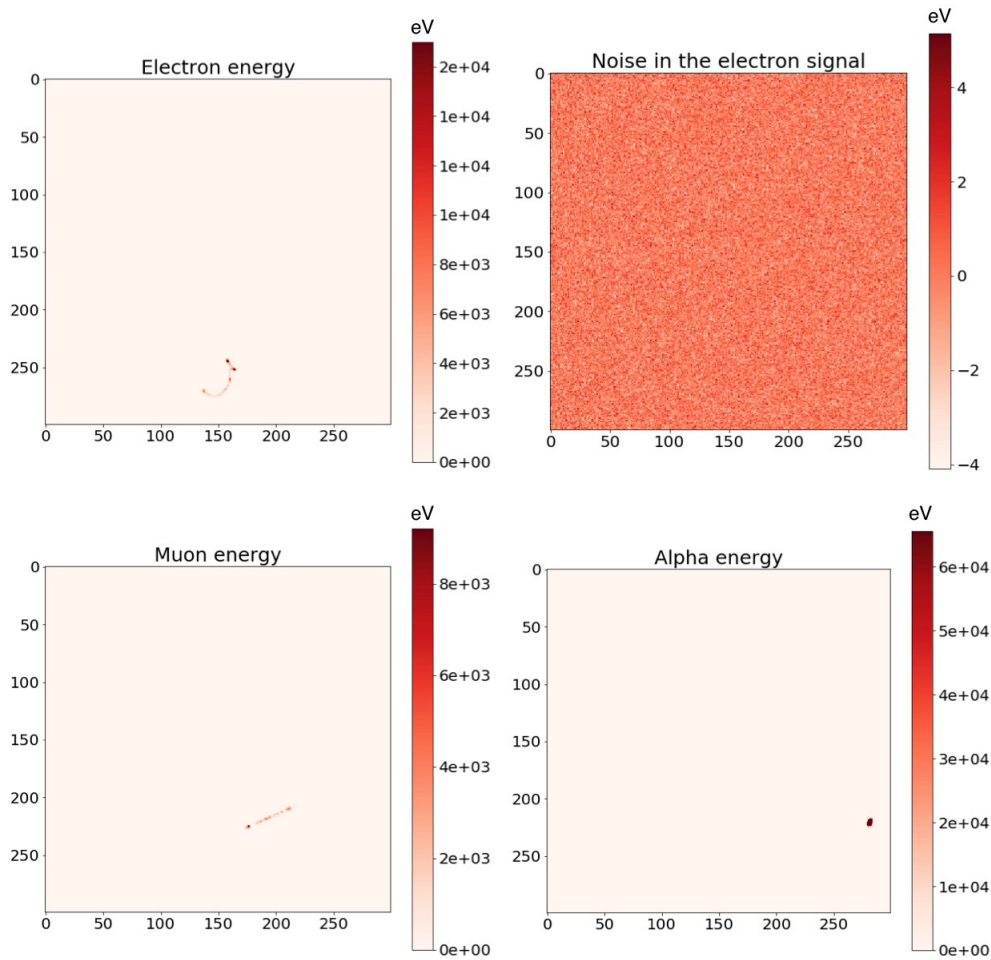


Figure 9: Electron signal with its corresponding noise in the first row, and alpha and muon signals in the second row

4.2 Data preprocessing

In order to reduce the number of parameters of the neural network to process, we will reduce the size of each energy and noise image. To do so, we shall follow the next process:

- Extract the particle track from the original energy image
- Calculate the size of the maximum track which will be called the shape of the *maximum window*
- Pad (randomly) with empty pixels all cropped energy images until they reach the size of the maximum window
- Crop the noise image to the shape of the maximum window

In Figure 10, the process just discussed, is shown. First we observe the original track of the electron and on the right, just the track. Finally, in the next row, both energy and noise have the size of the maximum window.

If we face a multiclass classification problem (with more than one type of particle), it is necessary to compare the resulting maximum windows and to choose the biggest, so we do not cut some tracks unintentionally.

In order to understand the preprocessing of the data of the three particles, the code can be found in reference [12]. There are three blocks of code, one for each particle. The first part of each block shows how the maximum window is created and how the energy images are cropped to extract the particle track.

Once the energy images are cropped and the maximum window has been calculated, the code available on reference [13] shows how the energy images are padded randomly and the noise is cropped to the shape of the maximum window. For this purpose, the first block of code displays two functions. The first one, `padding_energy`, pads randomly with empty pixels the energy image until its shape is the same as the maximum window. The second one, `cropping_noise`, just reshapes the noise image with the shape of the maximum window by selecting as many pixels as required. This is how the last row of images shown in Figure 10 is obtained.

The functions just discussed help us to preprocess the data, and as a result, we obtain energy images and noise images separately which will be the input for a convolutional neural network. This network will distinguish between each type of particles and noise.

On the other hand, we could also add the noise saved for each event to each energy image to obtain the real image. To do so, we would perform the sum of the noise and energy arrays as is described in reference [14]. In this case, the aim of the network would be the same as the one mentioned before: to classify between each type of particle and noise.

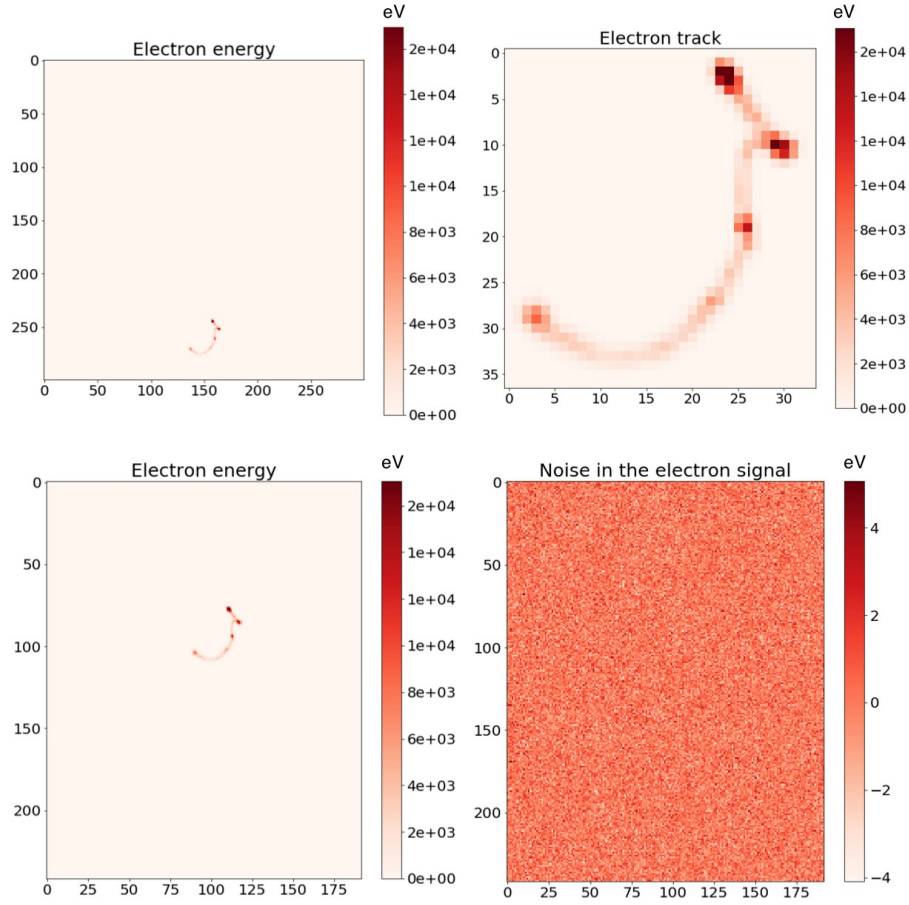


Figure 10: Data preprocessing for the energy, which is cropped and then padded, and for the noise, which is just cropped

Before feeding the neural network, it is important to perform the **standardization** of the data in order to standardize the range of features of input dataset [15]. Standardization is necessary when features of input dataset have large differences between their ranges. As this fact could influence the decision of the network, standardization is applied to avoid this scenario.

Once the data preprocessing has been discussed, it was not possible to load the entire dataset due to insufficient memory on the machine, however this problem is becoming increasingly common. In order to solve this memory issue, a **data generator** has been created (available in reference [16]). Data generators are iterators, therefore, when functions like *fit generator* or *predict generator* are called, they loop through the data as many times as necessary.

4.3 Classification and localization tasks

The first step to create the algorithm to **classify** different particles is to build a convolutional neural network capable of distinguishing between energy signals (without noise added) and noise. In order to do so, the network will have an output layer with four neurons, one for each class: electrons, muons, alphas and noise.

Then, we will add the noise stored to the energy data to check if the network is still capable of the classification in these four classes.

After these models are created, we can focus on simpler scenarios, such as a binary classification problem for electrons and muons, where we could also add the noise stored.

Once the classification tasks have been discussed, convolutional neural networks can be also used for the **location** of particles. It is noteworthy that for an image classification setting, data used is in the form (X, y) where X is the image and y are the class labels; however, in the of classification and localization, data will be used in the form (X, y) , where X is still the image and y is a array containing $(class\ label, x, y, h, w)$ where, x stands for bounding box bottom left corner x-coordinate, y for the bounding box bottom left corner y-coordinate, h will be the height of bounding box in pixel and w , the width of bounding box in pixel [17].

The number of output nodes will change as well. For the classification tasks, four nodes were needed (one for each particle and the last one for the noise) but now, seven nodes will compose the output layer: three for the particles, and the remaining, for location parameters just introduced. It must be emphasized that for the classification and localization problem, the noise will not constitute a class since it can not be located.

Finally, the loss function will be substituted by a linear combination of two loss functions, one for each problem, as following:

$$loss = \epsilon \rightarrow L_{class} + (1 - \epsilon) \rightarrow L_{loc} \quad (5)$$

where ϵ is a hyper-parameter that needs to be tuned since these two losses, L_{class} and L_{loc} , would be on a different scale.

For the classification problem, we will use a cross-entropy loss on top of the softmax activation (equation (6a) shows the loss function); and for the localization problem, the regression L2 loss function (equation (6b)).

$$L_{class} = - \sum_j y_j \log \hat{y}_j \quad (6a)$$

$$L_{loc} = \sum_j (\hat{y}_j - y_j)^2 \quad (6b)$$

where \log stands for the natural logarithm, \hat{y}_j is the network output, and y_j represents the true label.

5 Model details

In this section, we will discuss about the layers used to build both type of models: to classify, and to classify and locate the particles.

5.1 Models to classify

In order to analyze a model that classifies particles, we will use as an example the one that classifies the three types of particles, and the noise [18]. These models used for the classification problem are based on **sequential models**, which are appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor [19].

The input data for this model were the preprocessed images which were cropped to the shape of the maximum window calculated, and then standardized. Finally, we needed to add one last dimension to the numpy arrays: the number of channels, which in this case, would be 1 as it is not a RGB image. Hence the shape of the input layer is (window_y, window_x, 1). With this first input layer, we start the block of the convolutional 2D + max-pooling layers. Each convolutional layer would have a different number of filters (typically 32, 64 or 128) and the following max-pooling layer would have a sliding window of 2 × 2. All these layers have ReLU activation to ensure there is no vanishing gradient. After the convolutional block, we move on to the fully connected block through a flatten layer. Then, some dense layers were added with some dropout layers in between to avoid overfitting, to finally get to the dense output layer with softmax activation and four neurons to classify into the four categories. The schema below shows the architecture of the network:

```
model.add(layers.Conv2D(32, (3,3), activation='relu', input_shape=(window_y,
window_x, 1)))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(32, (3,3), activation='relu'))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(64, (3,3), activation='relu'))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(64, (3,3), activation='relu'))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(128, (2,2), activation='relu'))
model.add(layers.MaxPooling2D((2,2)))

model.add(layers.Flatten())
model.add(layers.Dropout((0.5)))
model.add(layers.Dense(units= 92, activation='relu'))
model.add(layers.Dropout((0.25)))
model.add(layers.Dense(units= 4, activation='softmax'))
```

The models to classify suffered from an **error** during the data preprocessing: as discussed, there is a need to standardize the data to ensure quick convergence of the optimization algorithm and to avoid extra small model weights for the purpose of numerical stability. This standardization was not well performed for the classification models because of the creation of the batches with the data generator. The correct procedure is the following:

- Calculate the mean of the training dataset.
- Calculate the standard deviation of the training dataset.
- Subtract the train mean and divide by the deviation of the training dataset, both train and test datasets.

But the mean and deviation of the training dataset was not determined, instead the mean/deviation changed for every batch as the mean/deviation of the batch was calculated and then subtracted/divided for each batch.

This mistake was not repeated for the classification+location problem.

5.2 Models to classify and locate

For the location+classification problem, we used the Keras **functional API**, which allowed us to create more flexible models as this API can handle models with non-linear topology, models with shared layers, and models with multiple inputs or outputs [20]. For these models, we focused on models with multiple outputs as we needed three output neurons to classify the particles and four others to locate the particle track within the CCD.

These models were based on convolutional neural networks, hence the first block of layers was always composed of filter + max-pooling (to reduce the number of parameters). Once the output feature map has decreased considerably, we would apply the flatten layer which prepares a vector for the fully connected layers. Afterward, the last fully connected layer would be split into two output layers, one to classify and the other, to locate. This schema can be visualized in Figure 11, where several blocks of filter + max-pooling were omitted.

As with the case of classification, the same amount of filters would be applied, and each block of filters would be followed by a max-pooling layer. All these layers have ReLU activation. After the convolutional block, we move on to the flatten layer and then to the fully connected block. Then, some dense layers were added to finally get to the two dense output layers. The schema below shows the architecture of the network:

```
inputs = keras.Input(shape=(window_y, window_x, 1))
x = layers.Conv2D(32, 3, activation='relu')(inputs)
x = layers.MaxPooling2D(2)(x)
```

```

x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.MaxPooling2D(2)(x)
x = layers.Conv2D(64, 3, activation='relu')(x)
x = layers.MaxPooling2D(2)(x)
x = layers.Conv2D(64, 3, activation='relu')(x)
x = layers.MaxPooling2D(2)(x)
x = layers.Conv2D(128, 3, activation='relu')(x)
block_output = layers.MaxPooling2D(2)(x)

out = layers.Flatten()(block_output)
x = layers.Dense(3002, activation='relu')(out)
output1 = layers.Dense(3, activation='softmax', name='output1')(x)
output2 = layers.Dense(4)(x)
output2 = layers.LeakyReLU(alpha=0.3, name='output2')(output2)
model = keras.Model(input=inputs, output=[output1, output2])

```

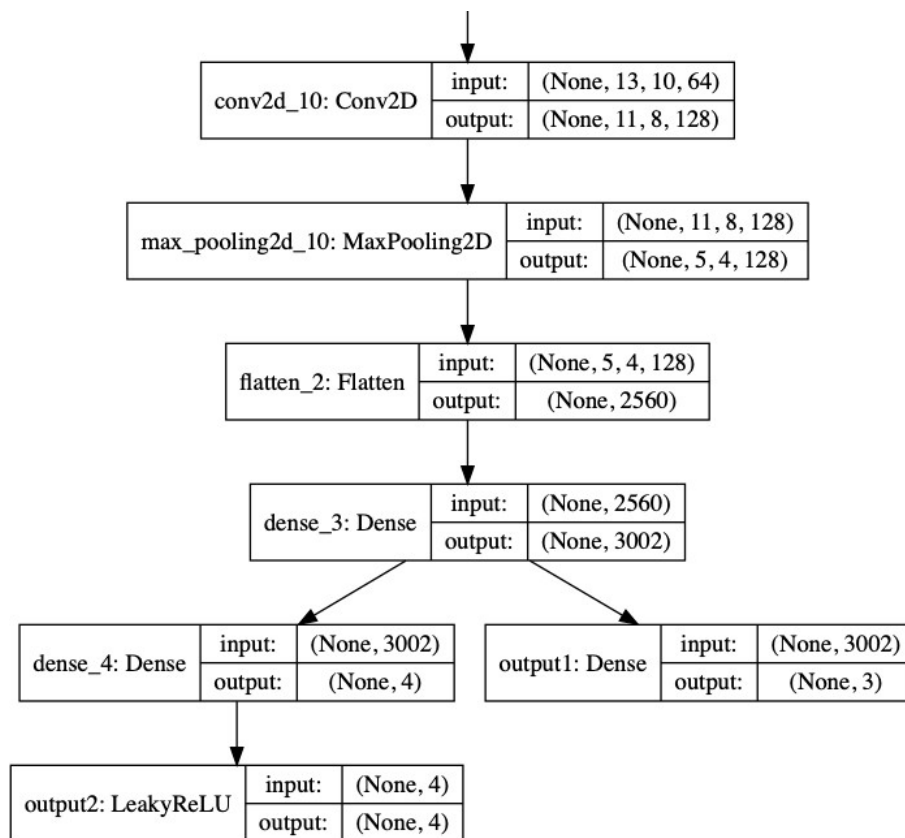


Figure 11: Final part of the model showing the flatter layer, the fully connected and the multiple outputs layers

As we can see, there are two output layers named after `output1` and `output2`. The first one mentioned is the responsible for the classification of the particles, as it has a **softmax activation**. Whereas the second one is the responsible for the location, as it has a variant of the ReLu activation: **Leaky ReLu** (equation 7). This new activation function is useful for this case as the standardized parameters³ of the ground-truth bounding boxes are small, and sometimes even negative (approximately from -1 to 1). Leaky ReLu has an hyper-parameter, α , which defaults to 0.3 and states the weight of negative values.

$$f(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \quad (7)$$

The first input layer with 3 neurons would classify into the three particles: electron, muon and alpha particle. And then, the remaining output layer would have 4 neurons to predict the value of the four parameters that determine the bounding box.

6 Training details

Once the architecture of the networks has been discussed, we will expose some details about the training of the models just mentioned in the previous section, such as the optimizers, the loss functions, the callbacks used for the early stopping of the training to avoid overfitting, etc.

6.1 Models to classify

The training for the models that classify the different particles (model of the reference [18]) is performed after their **compilation**, where we configure our model with losses and metrics. In our case, we would use categorical cross-entropy **loss** as we are facing a multiclass classification problem. The **metric** used would be the accuracy already explained in section 3.2.

In the model compilation, we might choose an optimizer as well. For our case, we will use the RMSprop **optimizer** (Root Mean Square Propagation), which is a variant of the Stochastic Gradient Descent. The RMSprop takes into account previous weight updates when computing the next weight update, rather than just looking at the current value of the gradients [5]. An important tuning parameter of the optimizer is the **learning rate** as it determines the step size at each iteration while moving toward a minimum of the loss function. In other words, the learning rate determines how big a step is taken in the direction the gradient is pointing at. It is important to note that a high value will cause the gradient jump over minima while a low value will cause the gradient get stuck at a local minimum. For our case, we usually chose a small learning rate. The code for the compilation is shown bellow.

³To check the standardization of these parameters and the input data, consult reference [21]

```
model.compile(optimizer=optimizers.RMSprop(lr= 1e-4),
              loss='categorical_crossentropy', metrics=['acc'])
```

As discussed in the theoretical section, **early stopping** of the training is a form of regularization used to avoid overfitting. For this reason, we implement callbacks with patience 7, where patience is the number of epochs with no improvement after which training will be stopped.

```
my_callbacks = [callbacks.EarlyStopping(patience=7)]
```

For this specific case, we chose the model to train for 40 **epochs**, although with the implementation of the callbacks, the number of epochs was reduced to 23. The next block of code shows how the model was trained: the training data was not just data but a data generator, same as the validation data. The number of steps per epoch corresponds to the number of the training batches (each of them containing 64 samples), same as the validation steps, which corresponds to the number of validation batches.

```
model.fit_generator(train_gen, epochs=40, validation_data=val_gen,
                   steps_per_epoch= 225, validation_steps= 75, callbacks=my_callbacks)
```

Each epoch lasted approximately 325 seconds, so the training lasted for more than 2 hours with a Intel(R) Core(TM) i7-4770HQ CPU @ 2.20GHz.

6.2 Models to classify and locate

The training for the models that classify and locate at the same time (model of the reference [22]) is performed after their compilation, as already mentioned.

For this case, we used a weighted linear combination of two **loss functions**: the categorical cross-entropy and the L2, where the first one is used for the classification and the other one, for the localization. For this linear combination, the hyper-parameter α took the value 0.855. Each loss has associated a **metric**: for the classification, we would use the accuracy again, but for the location, we would use the already introduced Intersection over Union (IoU).

For the model compilation, we chose the same **optimizer** as before: RMSprop. In this case, as before, we used a small learning rate. It is possible to observe in the following code the model compilation with the two different loss functions (categorical cross-entropy and L2) and their correspondent weights together with the metrics used for the two outputs, where `output1` corresponds to classification and `output2` corresponds to the localization task. Although the IoU could not be implemented during the training, it was used for the testing set to check the performance of the model.


```

losses={'output1': 'categorical_crossentropy', 'output2': 12}
alpha = 0.855
lossWeights = {"output1": alpha, "output2": 1-alpha}
model.compile(optimizer=optimizers.RMSprop(lr= 1e-4), loss= losses,
loss_weights= lossWeights, metrics=["acc"])

```

Early stopping was also implemented for these type of models. As opposed to the loss function, the structure of the callbacks is maintained, but this time we reduced the patience of the model to 6.

```

my_callbacks = [callbacks.EarlyStopping(monitor="val_loss", patience=6)]

```

In this case, we chose the model to train for 40 **epochs**, although due to the callbacks, the number of epochs was reduced to 33. Below, we show how the model was trained. As before, the number of steps per epoch corresponds to the number of the training batches and the validation steps corresponds to the number of validation batches. This time, we have less batches as the class noise has been eliminated.

```

model.fit_generator(train_gen, epochs=40, validation_data=val_gen,
steps_per_epoch= 171, validation_steps= 56, callbacks=my_callbacks)

```

Each epoch lasted approximately 28 seconds, hence the training of this model lasted for 14 minutes thanks to a Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz.

7 Results

Throughout this section, we will analyze the results obtained with the different models built for the following tasks: i) location and classification of electrons, muons and alphas; ii) noise added as a class: classification of electrons, muons, alphas and noise; iii) classification of electrons, muons and noise; and finally, iv) classification of electrons and muons. The first three tasks will also be discussed with noise added to the energy signals in this section.

7.1 Analysis of results without noise added

The following subsections, briefly summarize the results obtained for the models where the energy signals have no noise added.

7.1.1 Model to classify into electrons, muons, alphas and noise

The main goal of the models available on reference [18] is to classify as particle (electron, muon or alpha) or noise. And then, try to differentiate between each type of particle. As we can see in Figure 12, the main goal is accomplished, which means that the model is capable of

classifying correctly as noise or as particle 100 percent of the time. Furthermore, the model also concludes with a satisfactory classification of alpha particles. However, the model confronts some difficulty when attempting to distinguish between electrons and muons.

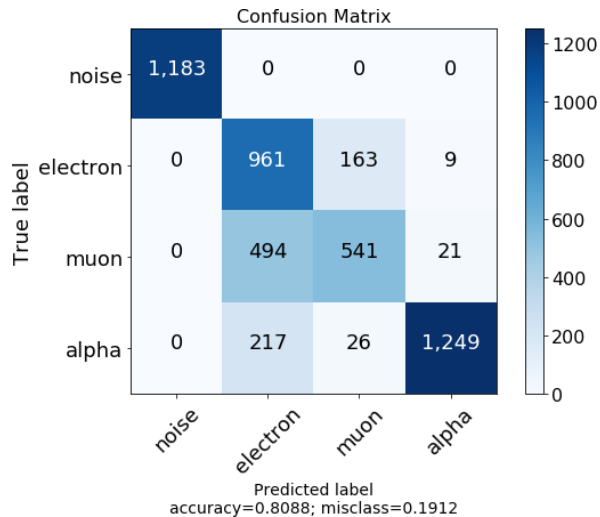


Figure 12: Confusion matrix for the model that classifies between noise, muons, alphas and electrons

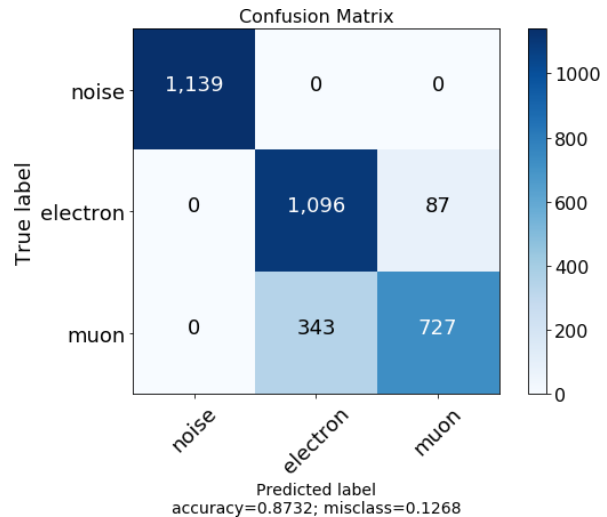


Figure 13: Confusion matrix for the model that classifies between noise, muons and electrons

7.1.2 Model to classify into electrons, muons and noise

The capacity to disentangle muon tracks from electron signatures fails in almost 50 per cent of the cases with the model presented in the previous section. Along this section, we will discuss a model built to distinguish between electrons, muons and noise, leaving aside alpha particles [23]. The confusion matrix (Figure 13) shows again that the noise signal are perfectly distinguished from energy signals, nevertheless there is still confusion between the two types of particles: electrons and muons. The model tends to classify as electron in case of confusion, but not backwards.

7.1.3 Model to classify into electrons and muons

Previous models find it hard to differentiate between muons and electrons, for this reason, a new model to distinguish between these two particles has been built [24] and the highest accuracy achieved is around 0.84 (Figure 14). In order to find out the reason why the model is not capable of distinguish both particles more accurately is analyzed below.

In Figure 15, the energy distribution was plotted to show whether the particle energy influenced the model to classify as muon or electron. Although, as we can see, the distribution of

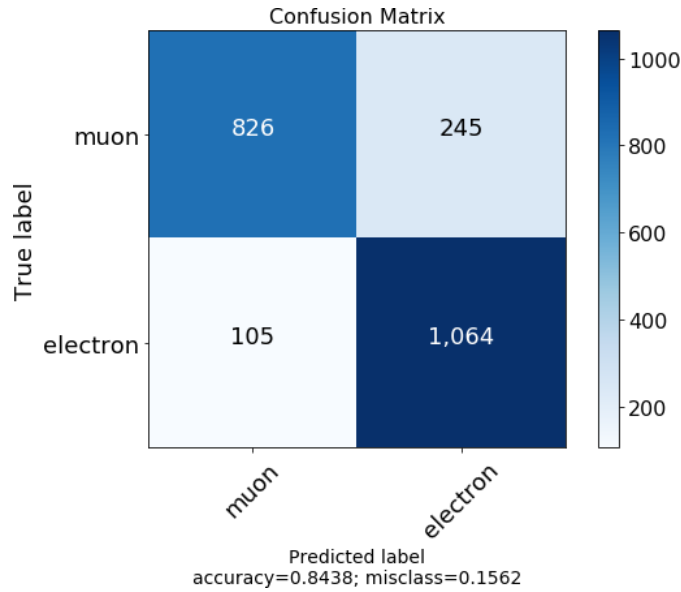


Figure 14: Confusion matrix for the model that classifies between muons and electrons

the misclassified particles overlaps the one for the correctly classified ones, hence the energy does not confuse the model.

Figure 16 displays the elongation distribution to show if there is a dependency on the length of the particle track. The elongation of each energy image was calculated as follows: the pixels that make up the track were located within a rectangle given by the coordinates of the bottom left and the top right corner of the track. Then, the number of pixels for each particle was calculated as the multiplication of the height and width of this rectangle.

Once the elongation distribution has been plotted, it is also a good idea to plot the error rate distribution (Figure 17) which shows that the elongation of the particle is not a determining factor in deciding whether to classify as electron or muon, because the error rate accumulates between 0 and 200 pixels, as the original pixel number distribution.

The elongation distribution is not able to show the shape of the poorly classified particles. For this purpose, the plot of Figure 18 was created. This plot allows us to ensure the fact that particles with both height and width small are the ones misclassified and lead us to confirm that the model cannot distinguish small and round tracks. In addition, to support this fact, we can see some mistaken tracks of both electron and muon in Figure 19 and some tracks of correctly classified electrons and muons in Figure 20. Hence, we can conclude that some tracks look alike and seems reasonable to think that the model could find hard to know which particle corresponds to the track, whereas the model classifies correctly if the tracks look different.

Finally, a better model could not be created and a physical reason for this particle confusion may lie in the fact that muon and electron signatures are not easily distinguishable at a glance. This issue may also be influenced by the z parameter (depth within CCD) responsible for the charge (energy) diffusion, which has not been taken into account directly. For instance, a muon track looks like an electron when arrives perpendicular to the CCD surface. In these cases, the electron diffusion is not playing an important role and the muon track can be mistaken for an electron.

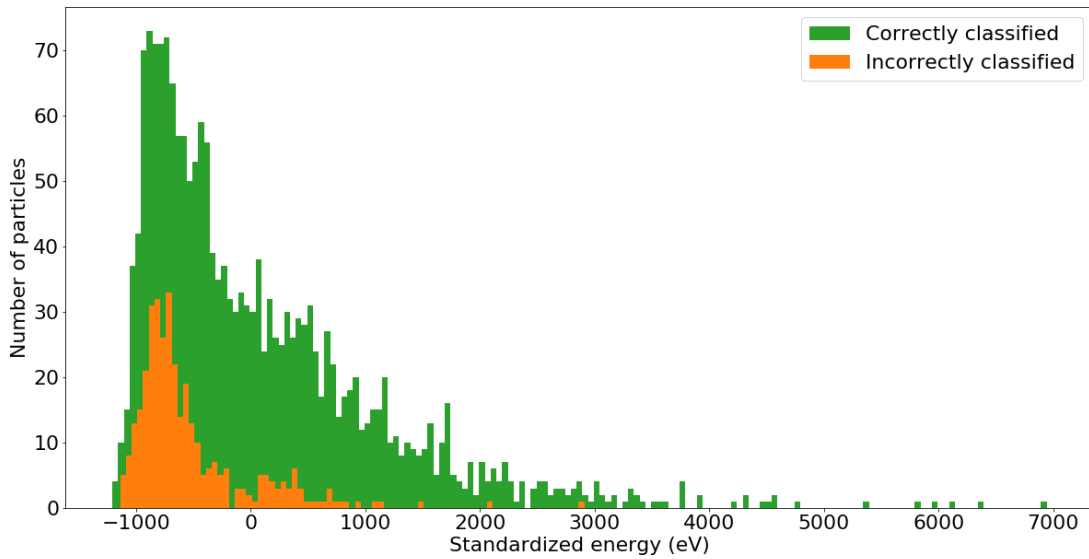


Figure 15: Energy distribution for the model that classifies between muons and electrons

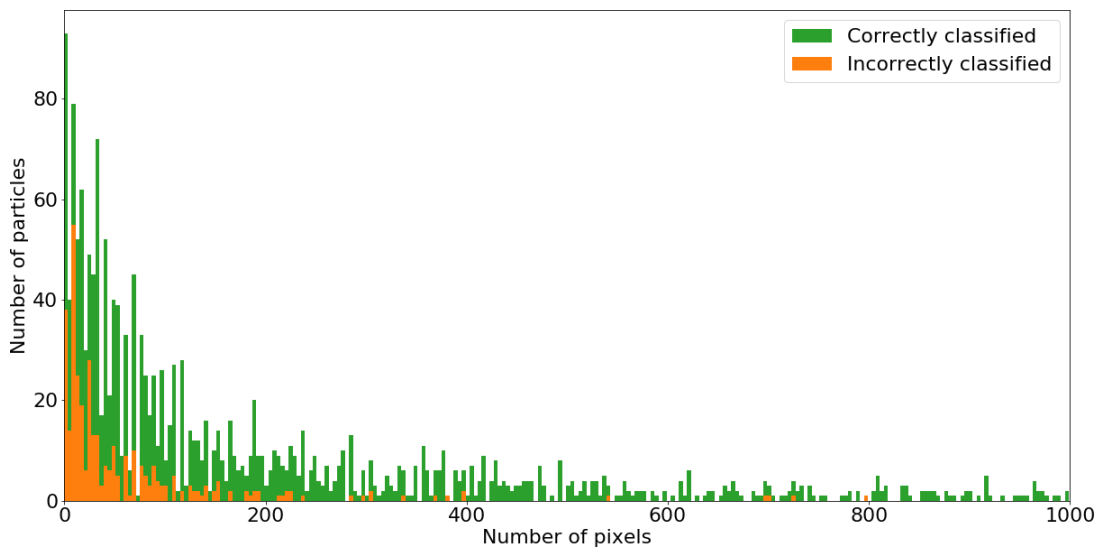


Figure 16: Elongation distribution for the model that classifies between muons and electrons

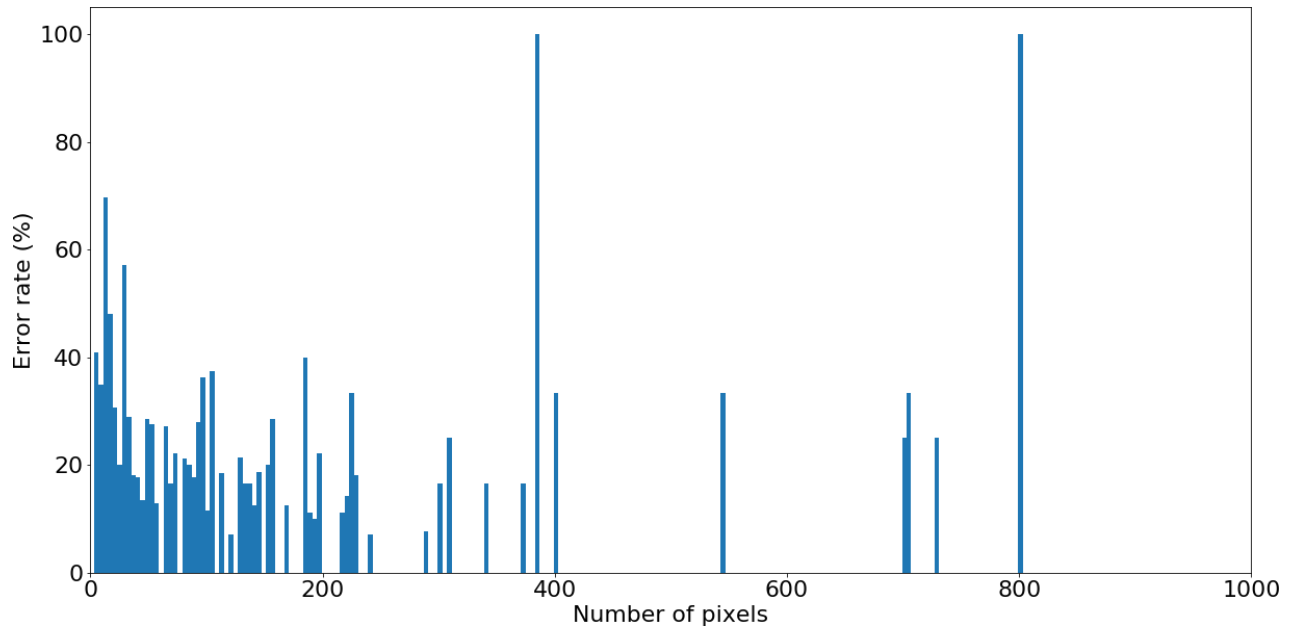


Figure 17: Error rate for the elongation distribution

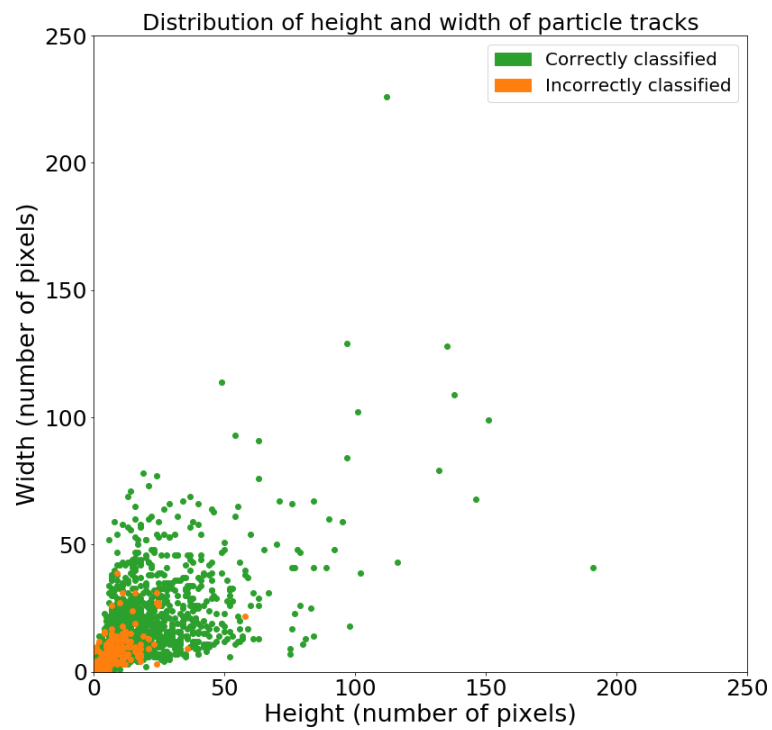


Figure 18: Height and width distribution for the model that classifies muons and electrons

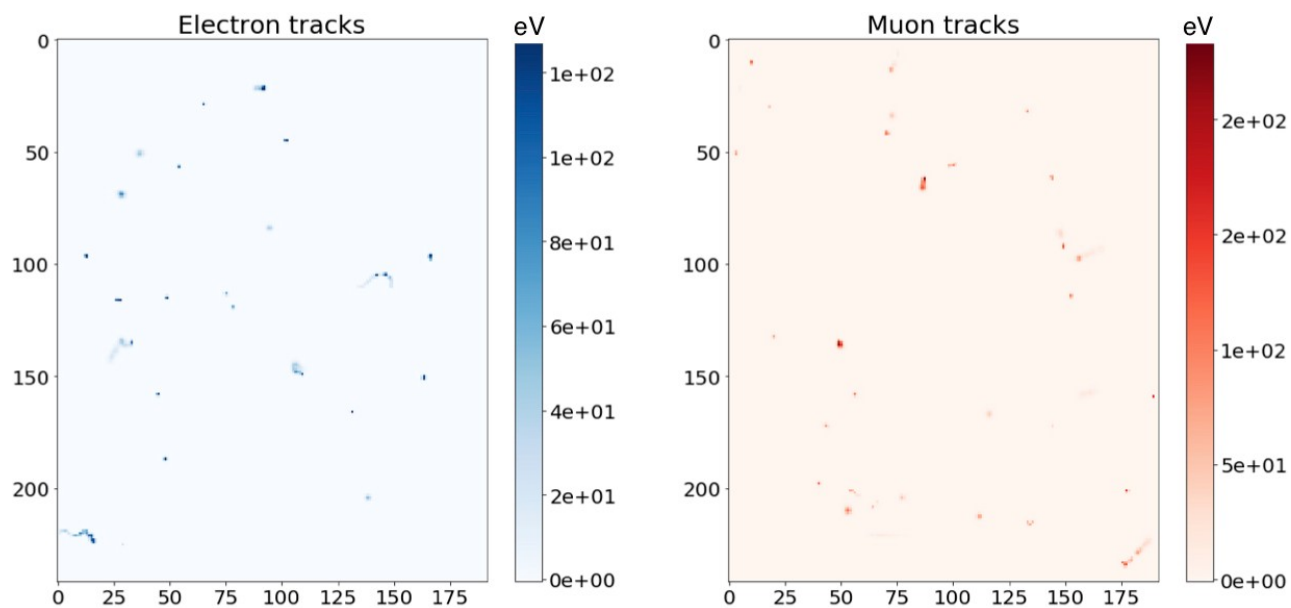


Figure 19: Mistaken electron and muon tracks

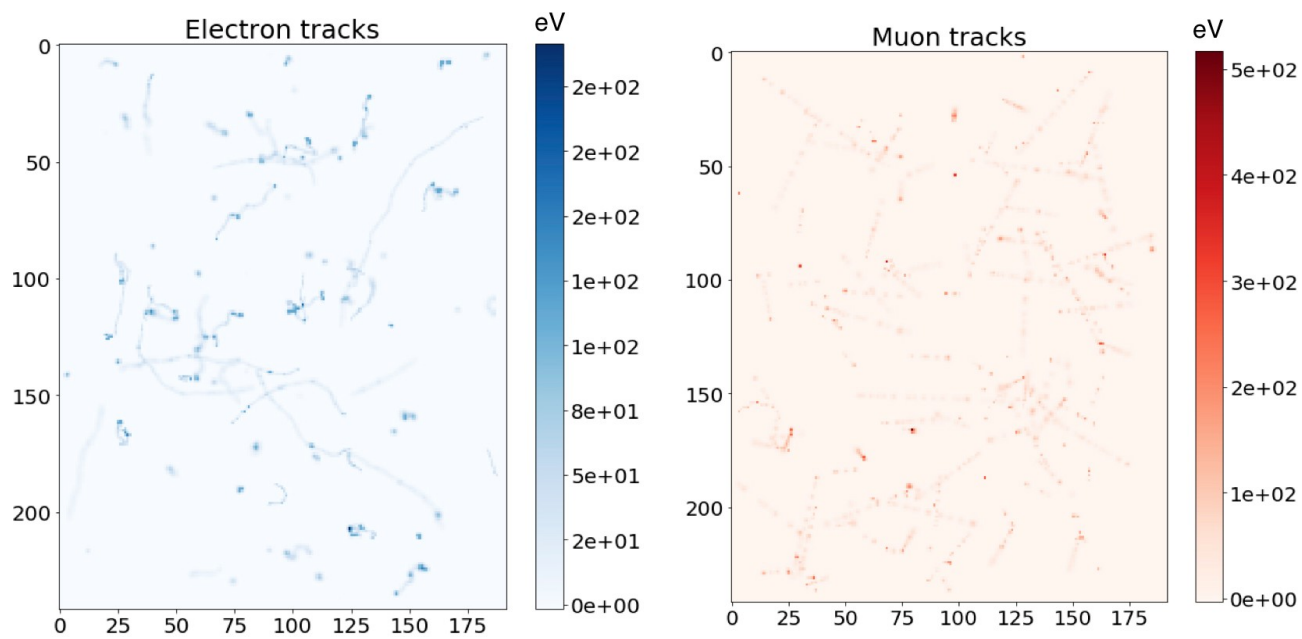


Figure 20: Tracks from correctly classified electrons and muons

7.1.4 Model to classify and locate electrons, muons and alphas

Once we studied the results of classification, we will add a location task to analyze the classification+location problem. The model analyzed below, corresponds to the best model reached in reference [22]. For the classification task, we can observe the results obtained in Figure 21. The confusion matrix shows again that alpha particles are well distinguished, this feature was also present in the models already discussed. This time, it was not possible to reach the same accuracy as before.

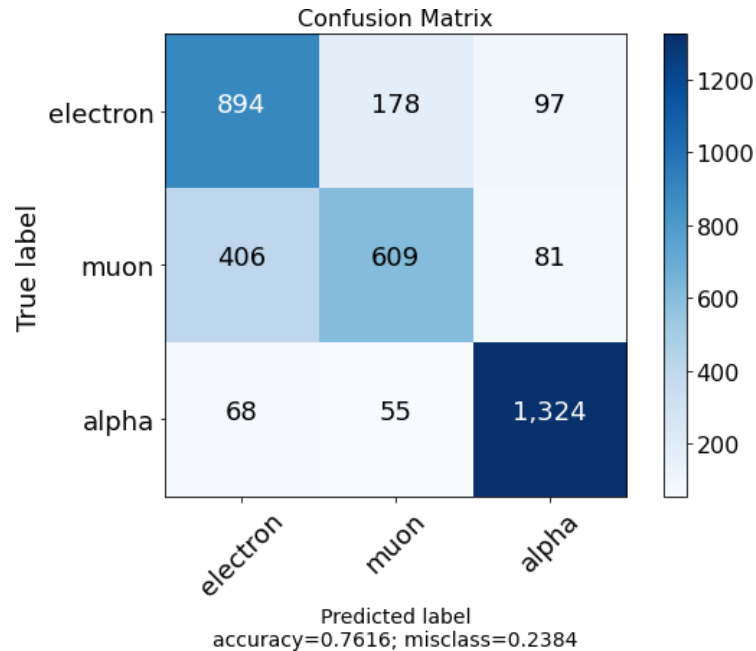


Figure 21: Confusion matrix for the model that classifies and locates particles

This model was also capable of localizing the particles with a 0.29 value of the IoU, which is a low value as a regular model would have 0.5 IoU value. Even though, the model can locate more or less the particle but not precisely.

In order to find a model that could both classify and locate particles was harder because the number of hyper-parameters to tune was larger. For these models it was greatly influencing the ϵ parameter and as before, the number of neurons and layers in the fully connected block.

When we were only facing the classification problem, we just worried about the number of neurons per layer and number of layers. But when the localization task was added, it was harder to find the correct parameters for both tasks.

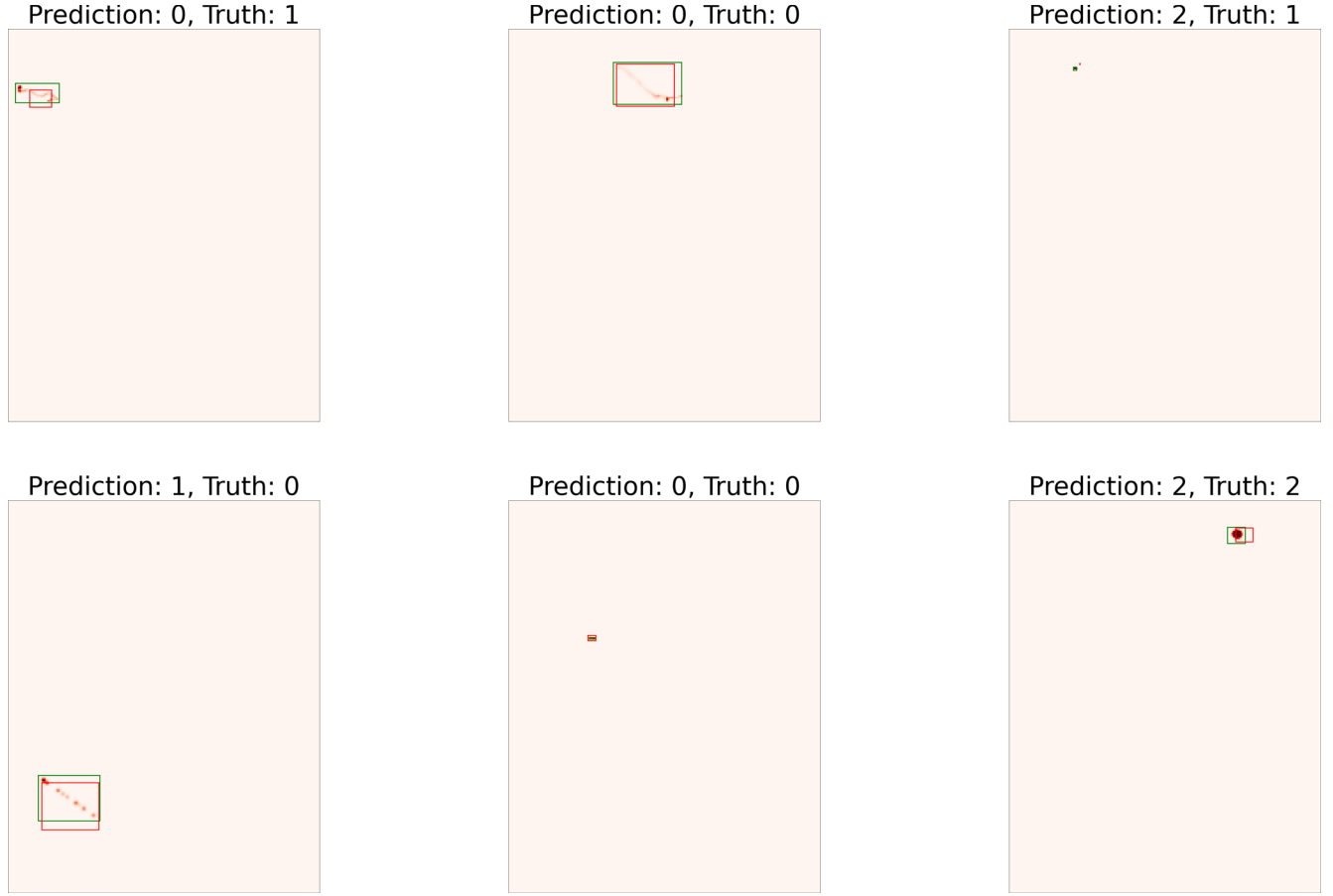


Figure 22: Localization and classification of the particles where 0 stands for electrons, 1 for muons and 2 for alpha particles

7.2 Analysis of signals with noise

Previously, we discussed the results where no noise was added to the energy, now we will proceed analogously, but in this case, the noise stored in the file will be added to the energy by adding the two numpy arrays.

7.2.1 Model to classify into electrons, muons, alphas and noise

Similarly, we will focus on the classification task for the particles and the noise. This time, as the noise has been added to the energy, the model (reference [25]) created is not capable of perfectly distinguishing between particles and noise (Figure 23). The recall for the noise is still 1.00 but this time, the precision drops to 0.84, whereas in the case without the noise added, both variables took the value of 1.00. The recall value means that all noise images are classified as noise when they are fed to the model; while the value of the precision means that there is a (100-84) percent chance that the model confuses a signal from a particle with a noise signal.

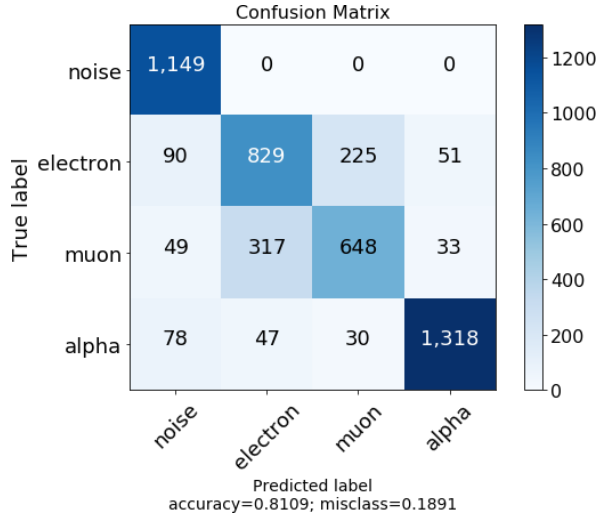


Figure 23: Confusion matrix for the model that classifies between noise, muons, alphas and electrons

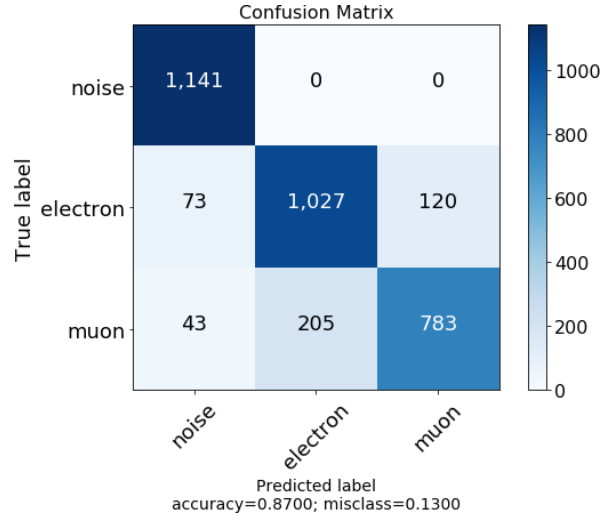


Figure 24: Confusion matrix for the model that classifies between noise, muons and electrons with noise added to the energy signals

7.2.2 Model to classify into electrons, muons and noise

As discussed in the section of the analysis for signals without noise, the noise and energy signals were perfectly classified. Now, we will study a more realistic scenario where the noise has been added to the energy signals [26]. In the confusion matrix of Figure 24, the accuracy is calculated. Comparing this accuracy with the one obtained without noise added (Figure 13), we can confirm that the accuracy decreases in this more realistic scenario. Even so, the results for other statistical values are presented in Table 2.

The recall for the noise is 1.00, this value means that if we feed the neural network with a noise signal, it will be classified as noise 100% of the time; while if the model receives an energy signal (electron or muon), it is capable of classifying as noise (100%-91%) of the time.

	precision	recall	f1-score	support
noise	0.91	1.00	0.95	1141
electron	0.83	0.84	0.84	1220
muon	0.87	0.76	0.81	1031
accuracy			0.87	3392
macro avg	0.87	0.87	0.87	3392

Table 2: Precision, recall, f1-score and support for the model of electrons, muons and noise with noise added

8 Conclusions

The enigma of dark matter is a problem of modern cosmology and particle physics. There are proposals for candidate particles to form dark matter, such as the WIMPs (*Weakly Interacting Massive Particles*). Our interest was focused on DAMIC-M. Thanks to Deep Learning techniques as the Convolutional Neural Networks, we have been able to study the background signals from Standard Model events. The main goal is to discard Standard Model events from what it could be a dark matter signal (if it really exists). For this reason, it is extremely important to establish a good working basis on these Standard Model signals.

Throughout this thesis, we have discussed different scenarios with three particles: muons, electrons and alpha particles; and faced two problems: localization and classification.

The classification task (with no noise added to the energy signals) showed good results when we were classifying as signal or noise, even though, the data standardization was not performed correctly. The model was always able to distinguish a particle track from noise, and successfully concluded the alpha particle classification. On the other hand, the model presented additional challenges when classifying as muon and electron. In order to improve the classification results of muons and electrons, this problem was studied in depth.

The tracks of both electrons and muons could result very similar if the particle impacts perpendicular to the CCD surface. In these cases, the electron confusion is low and hence, the muon tracks can be mistaken for electron tracks.

When the noise was added, the model was not capable anymore of distinguishing between noise and signal, but the accuracy did not drop abruptly. In fact, in this scenario, the network found easier to classify between the particles, and this is why the accuracy changes from 0.8088 to 0.8109, hence improving.

As soon as the location task was added to the classification task, the number of hyper parameters of the models increased. The most important parameters that could influence the behaviour of the network so far, were the learning rate of the optimization algorithm, the number of neurons per layer, and the number of layers. But now, the hyper parameter which defines the weight of the losses functions correspondent to the two tasks, becomes the most important one, as it will be a need to find the proper equilibrium between the two of them. Added to this parameter, we would also have the ones already mentioned such as the number of neurons or layers.

For the reasons just discussed, this multitasking model was unable to deliver as accurate results as the classification model, hence the maximum accuracy reached was around 76%. The location predicted was not precise either, since the IoU took the value of 0.29, which means that the network can locate the particle inside the image, but not accurately.

Finally, from the results just discussed, we can conclude that Convolutional Neural Networks are a powerful tool for particle classification due to their ability to learn spacial hierarchies and the fact that these learned patterns will be translation invariant.

9 Further work

The scenarios discussed deal with one particle track, but in a real scenario we could be facing a multi classification and location problem, even showing overlapping between particles.

Once the Standard Model events have been understood, the next step would be to dismiss these events as dark matter signals. We would study the events that were not classified as Standard Model events and the distortions of the intrinsic detect noise as the dark matter signal could mimic this effect. The input data for the models, now, would have a different noise distribution and probably, it would be necessary to add another class to classify as "distortion of the noise".

References

- [1] M. Habermehl, M. Berggren, and J. List, "WIMP Dark Matter at the International Linear Collider," 2020. [Online]. Available: <http://arxiv.org/abs/2001.03011>
- [2] G. Bertone, "Behind the scenes of the universe: from the Higgs to dark matter," *Choice Reviews Online*, 2014.
- [3] N. Castelló-Mor, "DAMIC-M experiment: Thick, silicon CCDs to search for light dark matter," *Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 2019. [Online]. Available: <https://arxiv.org/abs/2001.01476>
- [4] R. Essig, M. Fernández, F. Fernández-Serra, J. Mardon, A. Soto, T. Volansky, T.-T. Yu, and C. N. Yang, "Direct Detection of sub-GeV Dark Matter with Semiconductor Targets," Tech. Rep., 2016.
- [5] N. Ketkar, *Deep Learning with Python*, 2017.
- [6] E. Castillo, A. Cobo, J. M. Gutiérrez, and R. E. Pruneda, *Functional Networks with Applications*, 1999.
- [7] R. Cano, A. S. Cofiño, J. M. Gutiérrez, and C. M. Sordo, *Redes probabilísticas y neuronales en las ciencias atmosféricas*, 2004.
- [8] "vdumoulin/conv_arithmetic: A technical report on convolution arithmetic in the context of deep learning." [Online]. Available: https://github.com/vdumoulin/conv_arithmetic#convolution-animations

- [9] V. V. Asch, "Macro-and micro-averaged evaluation measures [[BASIC DRAFT]]," Tech. Rep., 2013.
- [10] A. Rosebrock, "Intersection over Union (IoU) for object detection - PyImageSearch." [Online]. Available: <https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/>
- [11] "numpy.savez — NumPy v1.17 Manual." [Online]. Available: <https://docs.scipy.org/doc/numpy/reference/generated/numpy.savez.html>
- [12] S. M. López Monzó, "TFM/electrons_vs_muons_vs_alphas_vs_noise/crop_particles.ipynb at master · silmaglo/TFM." [Online]. Available: https://github.com/silmaglo/TFM/blob/master/electrons_vs_muons_vs_alphas_vs_noise/crop_particles.ipynb
- [13] "TFM/data_eVSmuVSa.ipynb at master · silmaglo/TFM." [Online]. Available: https://github.com/silmaglo/TFM/blob/master/electrons_vs_muons_vs_alphas_vs_noise/data_eVSmuVSa.ipynb
- [14] "TFM/adding_noise.ipynb at master · silmaglo/TFM." [Online]. Available: https://github.com/silmaglo/TFM/blob/master/electrons_vs_muons_vs_alphas_vs_noise/adding_noise.ipynb
- [15] "TFM/standardizing_data(npz).ipynb at master · silmaglo/TFM." [Online]. Available: [https://github.com/silmaglo/TFM/blob/master/electrons_vs_muons_vs_alphas_vs_noise/standardizing_data\(npz\).ipynb](https://github.com/silmaglo/TFM/blob/master/electrons_vs_muons_vs_alphas_vs_noise/standardizing_data(npz).ipynb)
- [16] "TFM/data_generator.ipynb at master · silmaglo/TFM." [Online]. Available: https://github.com/silmaglo/TFM/blob/master/electrons_vs_muons_vs_alphas_vs_noise/data_generator.ipynb
- [17] R. Agarwal, "Object Detection: An End to End Theoretical Perspective." [Online]. Available: <https://towardsdatascience.com/object-detection-using-deep-learning-approaches-an-end-to-end-theoretical-perspective-4ca27eee8a9a>
- [18] "TFM/e VS mu VS a VS n.ipynb at master · silmaglo/TFM." [Online]. Available: https://github.com/silmaglo/TFM/blob/master/electrons_vs_muons_vs_alphas_vs_noise/e VS_mu VS_a VS_n.ipynb
- [19] "The Sequential model." [Online]. Available: https://keras.io/guides/sequential_model/
- [20] "The Functional API." [Online]. Available: https://keras.io/guides/functional_api/
- [21] "TFM/standardizing_data(npz).ipynb at master · silmaglo/TFM." [Online]. Available: [https://github.com/silmaglo/TFM/blob/master/electrons_vs_alphas_vs_muons_location/standardizing_data\(npz\).ipynb](https://github.com/silmaglo/TFM/blob/master/electrons_vs_alphas_vs_muons_location/standardizing_data(npz).ipynb)

- [22] “TFM/e VS mu VS a Loc.ipynb at master · silmaglo/TFM.” [Online]. Available: [https://github.com/silmaglo/TFM/blob/master/electrons vs alphas vs muons.location/e VS mu VS a Loc.ipynb](https://github.com/silmaglo/TFM/blob/master/electrons%20vs%20alphas%20vs%20muons/location/eVS_mu_VS_a_Loc.ipynb)
- [23] “TFM/e VS mu VS n 2.ipynb at master · silmaglo/TFM.” [Online]. Available: [https://github.com/silmaglo/TFM/blob/master/electrons vs muons vs noise/e VS mu VS n 2.ipynb](https://github.com/silmaglo/TFM/blob/master/electrons%20vs%20muons%20vs%20noise/eVS_mu_VS_n_2.ipynb)
- [24] “TFM/data eVS mu 2.ipynb at master · silmaglo/TFM.” [Online]. Available: [https://github.com/silmaglo/TFM/blob/master/electrons vs muons/data eVS mu 2.ipynb](https://github.com/silmaglo/TFM/blob/master/electrons%20vs%20muons/data%20eVS_mu_2.ipynb)
- [25] “TFM/e VS mu VS a VS n N.ipynb at master · silmaglo/TFM.” [Online]. Available: [https://github.com/silmaglo/TFM/blob/master/electrons vs muons vs alphas vs noise/e VS mu VS a VS n N.ipynb](https://github.com/silmaglo/TFM/blob/master/electrons%20vs%20muons%20vs%20alphas%20vs%20noise/e_VS_mu_VS_a_VS_n_N.ipynb)
- [26] “TFM/e VS mu VS n 2N.ipynb at master · silmaglo/TFM.” [Online]. Available: [https://github.com/silmaglo/TFM/blob/master/electrons vs muons vs noise/e VS mu VS n 2N.ipynb](https://github.com/silmaglo/TFM/blob/master/electrons%20vs%20muons%20vs%20noise/eVS_mu_VS_n_2N.ipynb)