Faculty

of

Science

Application of machine learning techniques to images collected with Charge Coupled Devices to search for Dark Matter

In partial fulfillment

of the Requirements for the Degree of

Bachelor of Science in Physics

Author: Aritz Lizoain Cotanda

Director: Rocío Vilar Cortabitarte

Co-director: Alicia Calderón Tazón

September 2020

Abstract

The Standard Model of particle physics, while being able to make accurate predictions, has been proved to fail to explain various phenomena, such as astronomical dark matter observations. In this work, a machine learning application has been implemented with the goal of studying dark matter candidates. Images from Charge Coupled Devices (CCDs) in different experiments DAMIC/DAMIC-M located underground will be used to test different deep learning algorithms. A U-Net model has been trained with Python's open-source library Keras. The model performs multi-class image segmentation in order to detect dark matter particle signals among background noise.

Key words

Machine learning, CNN, image segmentation, Python, DAMIC-M

Acknowledgements

First and foremost, I wish to express my sincere gratitude to my thesis director, Rocío Vilar Cortabitarte, and co-director, Alicia Calderón Tazón, for providing their expertise and guidance throughout the course of this project. I would also like to thank the rest of my thesis advisors, Agustín Lantero Barreda and Núria Castelló-Mor, who contributed so thoroughly through their assistance and dedicated involvement.

I take this opportunity to express gratitude to my friends, Asier, Ixaka, Santi and Toño, for joining me in the despair and feeling of accomplishment that studying physics can entail. Special appreciation to Kunkun, for all her love and loyal support. And my biggest thanks to my parents and sister for always believing in me and wanting the best for me.

Table of Contents

List of figures

List of Abbreviations

AI *Artificial Intelligence*

CCD *Charge-Coupled Device*

CNN *Convolutional Neural Network*

DAMIC *DArk Matter In CCDs*

DAMIC-M *DArk Matter In CCDs at Modane*

DM *Dark Matter*

ML *Machine Learning*

MP *Megapixel*

NN *Neural Network*

RGB *Red Green Blue*

1 Introduction

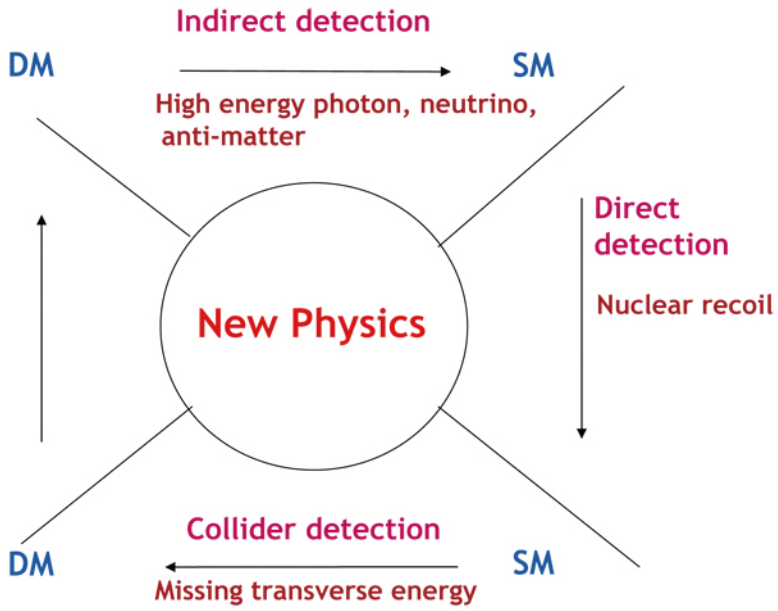
The problem of the DM in the universe, together with the explanation of dark energy, is still one of the major unsolved problems in cosmology, astrophysics and particle physics. Its identity links studies of the universe at both the largest and smallest observable scales. The DM problem, historically termed as the missing matter problem, was proposed by Fritz Zwicky in 1933, when his research on the Coma galaxy cluster inferred the existence of an anomaly. The virial theorem allows the calculation of galaxy masses making use of gravitational attraction. Zwicky calculated the theoretical galaxy cluster mass using the rotational velocity of luminous matter, and observed a discrepancy with the measured galaxy cluster mass. For the coming years various DM evidences were found, all of them based on gravitational interactions. However, given the universality of gravity, these evidences give very little information about the nature of DM.

The Planck mission's final data release (2018) showed that the dark energy density in the universe is $\Omega_\Lambda \approx 0.68$, while matter density is $\Omega_m h^2 \approx 0.32$, from which around 85% is non-baryonic. None of the Standard Model particles is a good DM candidate; most of them are unstable, with lifetimes far shorter than the age of the universe, and the rest contribute to the baryonic energy density Ω_b, which is too small to be considered as a possibility. A vast majority of the scientific community believes that the evidence for DM requires particles beyond the Standard Model. Neutralinos, gravitinos, sterile neutrinos, axions and other particles related to hidden dark sectors, such as dark photons, are some

of the researched candidates. Lately non-fundamental particles such as SIMPS (strongly interacting massive particles), and macroscopic objects such as primordial black holes, have also been proposed as DM candidates. Nonetheless, a minor part of the community does not believe in DM; they believe that the theory of gravitation is not complete, and work with alternative theories of gravity, such as Modified Newtonian dynamics (MOND).

Three strategies are followed for DM detection: accelerators, direct and indirect detection (see Figure 1).

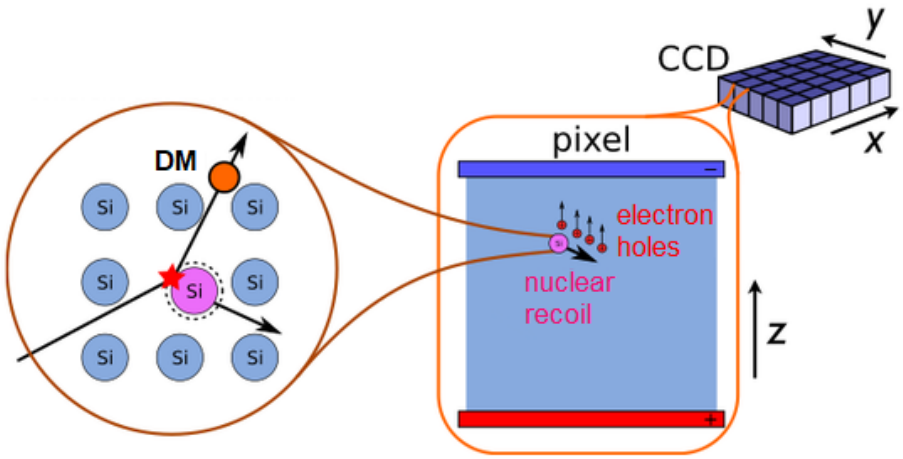
Figure 1. Sketch of different types of search strategies for DM detection. Digital Image. Virdee, T. S. *Beyond the standard model of particle physics*. (Royal Society, 2016). https://royalsocietypublishing.org/doi/10.1098/rsta.2015.0259.



1.1 DAMIC-M

In this study the direct search strategy is followed. It is a huge endeavor to develop experiments able to directly investigate the particle nature of DM. These experiments aim to identify recoils produced by the scattering between the theoretical DM particles and a detector's target nuclei or electron. Specifically in this research, the silicon of the CCDs is used as a target (see Figure 2). This kind of experiments are very sensitive to any radiation background, either from the construction material or cosmogenic. Moreover, the collision signals are expected to be rare and low (keV scale and below). In order to screen out the radiation background, the material is thoroughly assayed. In addition, carrying out the measurements in a subterranean location, inside a mountain or a mine, shielded from cosmic-rays induced events, is key to achieve sensitivity to DM particle detection.

Figure 2. Nuclear recoil produced by the scattering between a DM particle and a detector's Si nucleus. Digital image. Aguilar-Arevalo, A. et al. *Measurement of radioactive contamination in the CCD's of the DAMIC experiment.* (Journal of Physics, 2015). https://arxiv.org/pdf/1506.02562.pdf.



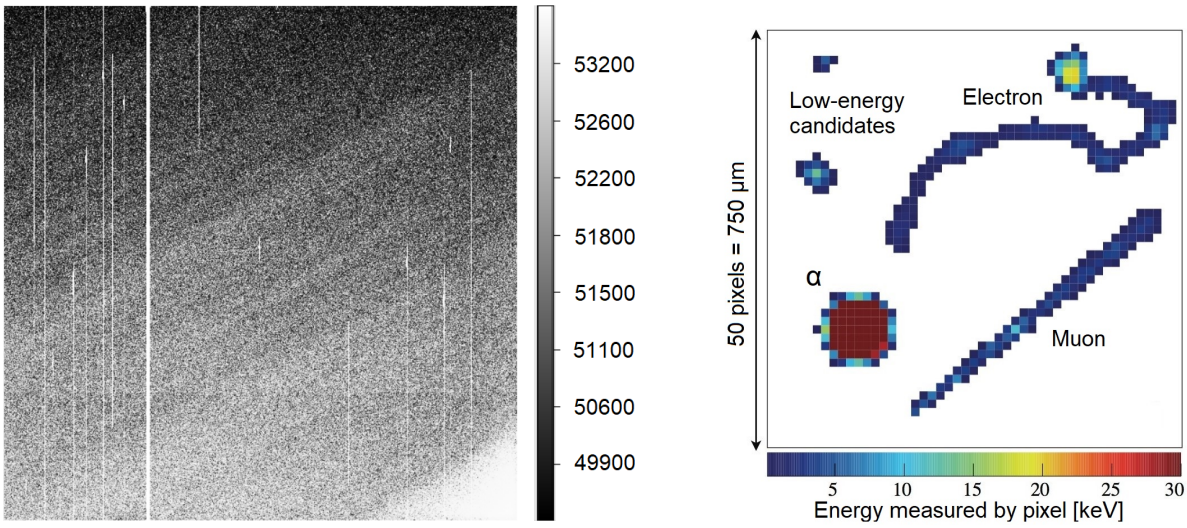
Previously calibrated and tested 675 μm-thick (approx. 15g each) CCDs are located inside an electroformed copper box, used for screening purposes. The 36MP CCDs have a very low radiation background (0.1/event/kg/day/keV) and a resolution better than $1e^-$ by means of the skipper readout system. The operating temperature can range between 135-140K and a maximum of 240K. An array of 50 skipper CCDs will constitute the future DAMIC-M experiment located in the Laboratoire Souterrain de Mondane facility (France), which is still finalizing its design and is scheduled to be installed by the end of 2023. Nevertheless, the collaboration has been studying this technology approximately since 2012 and data has been taken from the DAMIC experiment located at SNOLAB (Canada). The data from DAMIC is the one that has been used in this project.

CCD images contain a high-resolution two-dimensional projection on the XY plane of the charge deposits in the active volume of the device. The DAMIC data has been acquired with two different readout configurations: 1X1 and 1X100. The first one is the standard CCD readout, reading each pixel individually. On the second one instead, columns of 100 pixel rows are read individually. The image readout times are 24h and 8h, respectively for the 1X1 and 1X100 setups. Immediately after taking the image, a "blank" image is acquired, whose exposure is only a few seconds. Since the occurrence of a physical event during each readout mode is $< 5\%$ and $< 0.1\%$ (respectively for 1X1 and 1X100), most blank images contain only the image noise. The total exposure time presents a statistically consistent white noise distribution.

Different ionizing particles in a CCD include: straight track-shaped cos-

mic ray muons, large drop-shaped alpha particles, "worm"-shaped straggling electrons, and low-energy candidates, characterized by small round clusters (see Figure 3 *Right*). Furthermore, the ionizing particles need to be distinguished from noise signals such as hot pixels (i.e. pixels which look much brighter than they should), glowing and any other issue with the pixels. The major research problem is the difficulty involved in correctly masking out the background noise; signals from ionizing particles are almost at the same energy level as the background (see Figure 3 *Left*). In order to face the challenge of discriminating the different ionizing particles from the background noise, machine learning is proposed as a solution.

Figure 3. *Left:* DAMIC 4096×4096 pixel CCD image (T=240K). The pixel intensity values are given in ADCs[1]. *Right:* Signatures of different ionizing particles in a CCD (processed image). Adapted Digital image. Aguilar-Arevalo, A. et al. *Measurement of radioactive contamination in the CCD's of the DAMIC experiment.* (Journal of Physics, 2015). https://arxiv.org/pdf/1506.02562.pdf.



The goal of this project is to implement an innovative deep learning application able to extract all the information from the detector images. An

[1]Unit of measurement for charge in count of ADC through the digital output of the A/D converter.

automated quality monitoring system is sought with the purpose of identifying the main defects associated to the detector. Four main categories are discriminated on each image: background, glowing, hot pixels and pixel clusters. The ML algorithm implemented on the images performs a behavior generalization seeking to uncover the signal of each category. Thus, the output shows all the practical information at a glance; an ideal segmented image is displayed making each category clearly distinguishable. As a result, pixel clusters can be differentiated, leading to further research.

1.2 Simulated CCD images

In order to develop a ML model, data is vital. Frequently, collecting suitable data is more troublesome than writing algorithms. Depending on the sophistication of the problem, the number of parameters and the amount of data needed varies significantly. Most commonly used datasets for computer vision can provide over 100000 images, sometimes even a million of them. However, possessing such a vast and appropriate dataset is not always possible. Occasionally, data collection is overly costly, and the lack of data limits the model.

This obstacle is overcome creating images that simulate the ones taken by CCDs. The aim is to emulate the signatures of the four main categories to discriminate, so that the model will learn and subsequently be able to predict them on real detector images. To create a simulated image, first the background is defined by a fixed intensity value, or pedestal, to which an amount of noise can be added. Then the other three main

categories to discriminate are added to the image: glowing, hot pixels and pixel clusters. These three differ in shape and pixel intensity[2] (being pixel clusters the least intense, and glowing and hot pixels the most intense). The freedom of choice is the main advantage of this solution; it is possible to vary which objects (or categories) are included in a picture, the amount of them, their pixel intensities, size, position, etc. This way, a wide variety of simulated images can be used, ensuring that the model learns all kind of signals that can appear in a real CCD image.

[2]Pixel intensity is proportional to the collected charge by each pixel.

2 Machine Learning

> *Field of study that gives the ability to the computer to self-learn without being explicitly programmed.*
>
> *Arthur L. Samuel, 1959*

As data growth continues to escalate, algorithms must keep improving to be able to understand it all with higher speed and higher accuracy. Processing the high amount of data that is produced every day is becoming more challenging without the assistance of ML. ML is an AI subset which is focused on developing programs that are able to teach themselves to make accurate predictions when exposed to new data. It is found in diverse sectors, due to its wide usage in image recognition, speech recognition, medical diagnoses, trading, etc. There are three main types of ML: supervised, unsupervised and reinforcement learning.

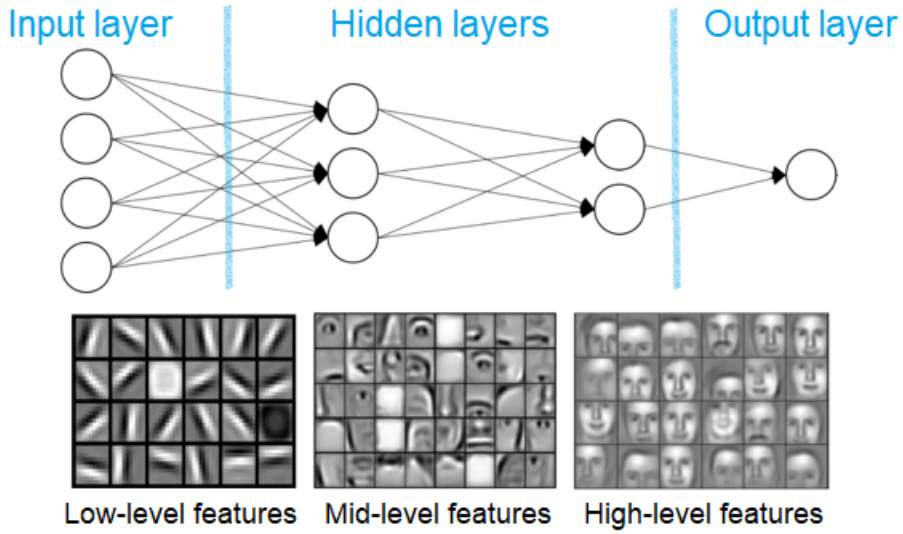
In supervised learning, algorithms are trained with labeled data. Thus, known input and output is being used. Each image data is tagged with its corresponding label, which is the desired output. The algorithm learns by comparing its prediction with the given label, and modifies the model accordingly. In this project classification supervised learning is applied, meaning that the output variables are categories ('background', 'glowing', 'hot pixels', and 'pixel clusters').

2.1 Convolutional Neural Networks

Taking the human brain as a reference, artificial NNs are based on connected nodes called neurons. A neuron receives inputs from other neurons and combines them together. The output from a neuron is obtained by a value transformation called activation function. The values used in the activation function, termed weights, are randomly initialized. The accuracy of a neuron output is determined by the loss function; the lower the loss, the higher the output accuracy. Therefore the goal is to find the right weights to minimize the loss function, thus, giving the most accurate prediction. This is done by an optimizer, that identifies which weights contribute most directly to the loss of the network, subsequently updating them in order to minimize such loss. The training process is repeated for a number of iterations, aiming to improve the weight value readjustment.

CNNs are composed of an input layer, several hidden layers, and an output layer. Their employment allows the recognition of specific properties of image data, thereby becoming highly suitable for computer vision applications. Images are passed through the NN as an array of values describing pixel intensities. Each of these values is a feature that characterizes the image. The first few neuron layers learn low-level features (basic elements such as edges and colors), leading to a more complex pattern learning by the succeeding layers (see Figure 4). This way, the network is able to differentiate one image from another. Generally, prediction accuracy is improved with a deeper network, i.e. with more layers.

Figure 4. CNN feature learning process. Adapted Digital Image. Torres, J. *Convolutional Neural Networks for Beginners.* (Towards Data Science, 2018). https://towardsdatascience.com/convolutional-neural-networks-for-beginners-practical-guide-with-python-and-keras-dc688ea90dca.



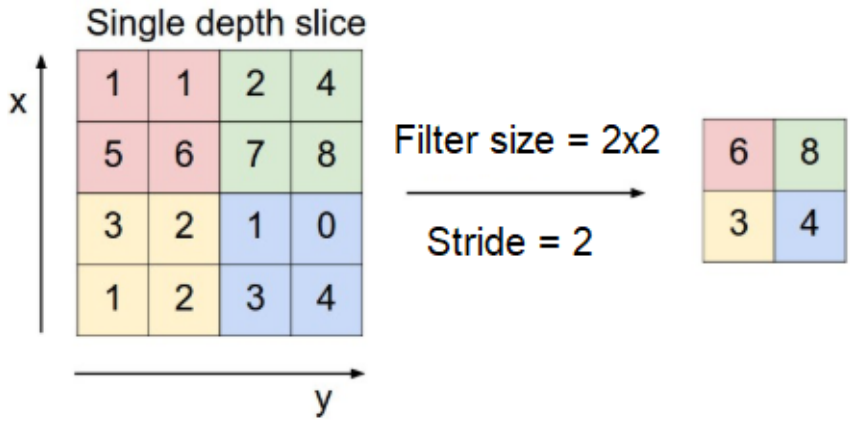
2.1.1 Convolution

CNNs are named after its most important layer, the convolution layer. While a standard NN layer applies its activation function weights to the whole image, a convolution layer applies a set of weights spatially across the image, thereby reducing the number of parameters needed. This set of activation function weights compose the filter, which is defined by several hyper-parameters: filter size, stride, and depth. Filter size sets the width and height of the filter. The number of pixels to move before applying the filter again is set by stride. If the stride is smaller than the filter size, regions of the image are overlapped. The depth defines the number of channels of the filter, which is equal to the number of input channels (e.g. for a RGB image the depth is 3, one for each color channel, while for a grayscale image the depth is 1).

2.1.2 Max pooling

Max pooling layers, like convolution layers, apply a filter across the image, which is also defined by a filter size and stride. The layer takes the maximum value within the filter, reducing the spatial size of the input (see Figure 5). However, it does not take the maximum value across different depths, since it is applied to each depth channel individually.

Figure 5. Max pooling operation example. Adapted Digital Image. CS231n Convolutional Neural Networks for Visual Recognition. https://cs231n.github.io/convolutional-networks/#pool.



2.1.3 Dropout

Overfitting is one of the most common issues when training a ML model. It causes the model to memorize the training data, instead of learning from it, which leads to a high accuracy on the predictions while training, but a low accuracy on testing predictions. The most effective solution is adding more training data. Nonetheless, adding a dropout layer also helps avoiding the issue. Dropout layers randomly ignore a number of neuron outputs, reducing the dependency on the training set.

2.1.4 Fully connected

A fully connected layer is usually added as the last layer of the CNN. Like in a standard NN layer, every neuron is connected to every neuron in the previous layer. The fully connected layer classifies the image based on the outputs of the preceding layers.

2.2 Image segmentation

An image classification problem consists in predicting the object within the image. On the other hand, image segmentation requires a higher understanding of the image; the algorithm is expected to classify each pixel in the image. Thus, the output is a labeled image in which each pixel is classified to its corresponding category. Self-driving cars development, medical images diagnosis and satellite image analysis are some of the numerous image segmentation applications.

2.2.1 U-Net

It is a widely held belief that a successful deep network training requires thousands of labeled training data. However, the U-Net network architecture, originally developed for biomedical (cell) image processing[3], uses the available labeled images more efficiently. In consequence, it has become one of the most popular networks in the medical domain, where

[3]Ronneberger, O. et al. *Convolutional Networks for Biomedical Image Segmentation.* (2015). https://arxiv.org/pdf/1505.04597.pdf.

usually thousands of training samples are beyond reach. Furthermore, the U-Net architecture is able to detect small size objects within the image.

The U-Net architecture contains two paths: the contraction path, called encoder, and its symmetric expanding path, the decoder (see Figure 12). The encoder is built stacking convolutional and max pooling layers. This way the size of the image is reduced, which is called down sampling. The deeper the network, the more reduced is the image. This allows obtaining information about the objects within the image, but the spatial information is lost. Therefore the image needs to be up sampled to the original image size, i.e. restore the low resolution image to a high resolution image. For this purpose, the decoder is built stacking convolutional and transposed convolutional layers. Every step of the decoder uses skip connections by concatenating the outputs of the down sampling layers with the up sampling layer at the corresponding level. The network does not contain fully connected layers, therefore is defined as a fully convolutional network.

3 Implementation

In this section the core of the project is dissected. Every employed method is explained; from the origination of an image, to the training of the model, every necessary step in the creation of the deep learning application is analyzed.

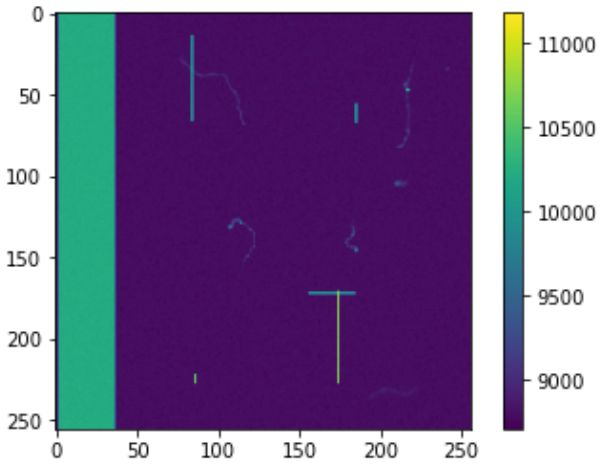
3.1 Image simulation

As stated in previous sections, the simulated images contained four classes[4] to segment: background, glowing, hot pixels and pixel clusters. Two datasets were created: a training and a testing set. The training set was used to train the ML model, while the testing set was only used once the model was fully trained. Trying to recreate real DAMIC images at T =140K as accurate as possible, specific pixel intensities were assigned to each class in 256×256 pixel images (see Figure 6).

The background intensity, also referred as pedestal, was set a value of 8800ADC with a noise of ±60ADC, meaning that the background pixel intensities took values between 8740 and 8860[5]ADC. Glowing was added as a vertical column with an intensity above background of range [1400, 1500]ADC. Hot pixels were added as thin vertical and horizontal lines of different lengths with intensities above background of 2200ADC, being the class with the highest pixel intensity value. Clusters, on the other

[4]Previously defined as categories.
[5]All pixel intensity ranges followed a gaussian distribution.

Figure 6. Simulated 256×256 pixel CCD image containing glowing, hot pixels and pixel clusters. The pixel intensity values are given in ADCs.



hand, were added from a file containing the intensity and position of 803 pixel clusters. The file did not contain any alpha particle, hence almost[6] all clusters were low-energy events and their intensity value was slightly above background.

Both python files containing the code for creating the simulated images, as well as the file with cluster information were provided by Agustín Lantero Barreda, PhD Student of DAMIC-M.

3.2 Image labeling

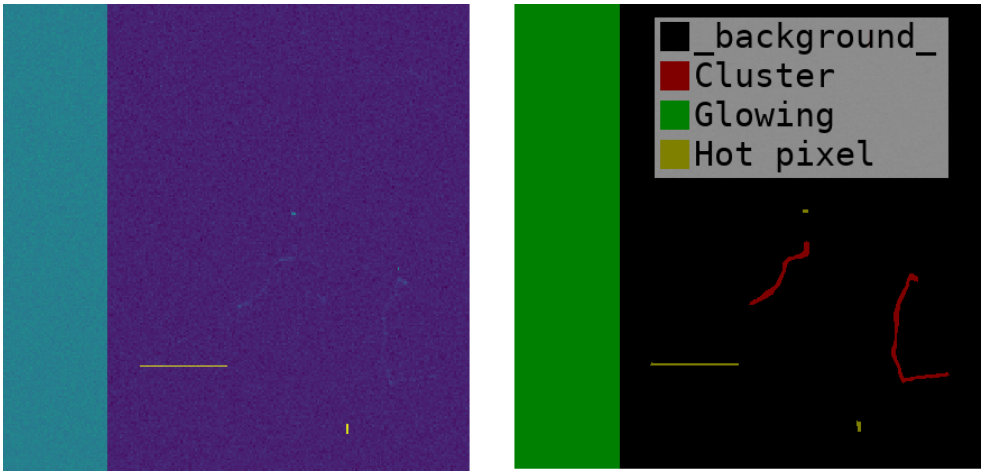
In supervised learning image labels are as essential as images. A label represents the desired output, i.e. an image where each pixel is classified to its corresponding class. A model learns by comparing its prediction with the label and trying to minimize the number of incorrectly predicted pixels.

[6]Some pixels representing electrons reached higher intensity values.

3.2.1 *Labelme*

Taking advantage of the existence of data labeling platforms, the graphical annotation tool *labelme* was used as a first attempt to create labels. Opening the simulated images in the graphical interface, each object was individually classified; the border of each glowing, hot pixel or cluster was drawn one by one with the computer mouse. Then, a JSON (JavaScript Object Notation) file was created and subsequently converted to PNG (Portable Network Graphics) file format. The created label contained each class differently colored (see Figure 7).

Figure 7. *Left:* simulated image. *Right:* image label created with *labelme*



Nonetheless, this method happened to be highly inefficient. On the one hand, individually drawing the border of each object within every image was very time-consuming. On the other hand, it was impossible to achieve an accurate label, particularly for clusters, due to the difficulty to distinguish them. For these reasons, a new labeling method had to be researched.

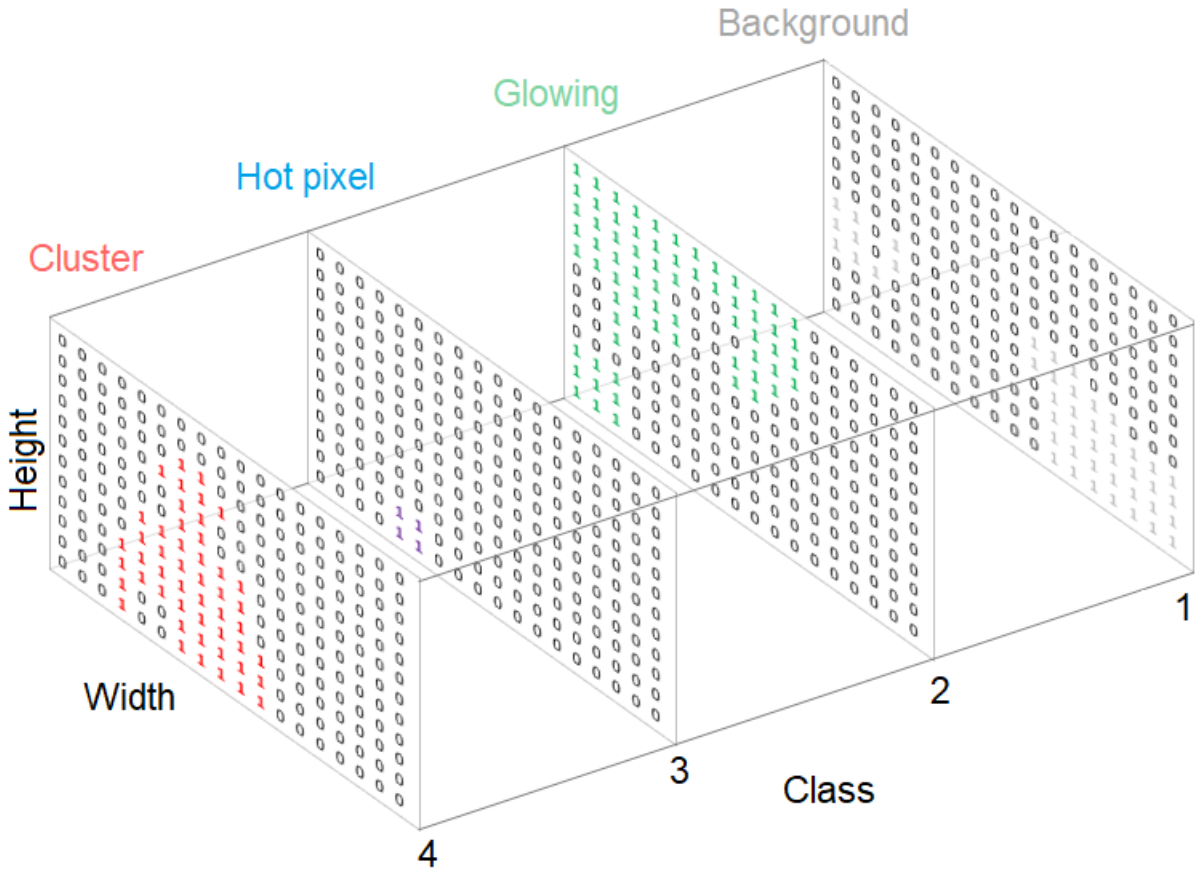
3.2.2 Automated labeling

A new method of image labeling was studied, straightforwardly without the employment of any labeling platform or annotation tool. Reflecting on how the simulated images were created, a great convenience was found: the images were created with predetermined pixel intensities. Thus, the threshold values that determined whether a pixel belonged to a certain class were known. This knowledge played a decisive part in the development of an automated labeling method.

The datasets were three-dimensional arrays of shape (number of images, height, width). The goal was to obtain a four-dimensional array of shape (number of images, height, width, 4), where the last dimension informed about the class corresponding to the pixel. This way, the label was a segmentation map where each pixel contained a class label represented as an integer; numbers 1, 2, 3 and 4 were respectively assigned to the classes 'background', 'glowing', 'hot pixel' and 'cluster'. These class labels were described in a one-hot encoded way, meaning that each one had a depth channel (e.g. [0,0,1,0] represented the class 'hot pixel').

To get started, a blank four-dimensional array was created. Then, going through all the images, each pixel was classified. Knowing the threshold values of the four categories, the pixel intensity value determined its class label. Based on this classification, the corresponding depth channel took value 1 while the other three remained at value 0 (see Figure 8). Going through all the images, the segmentation maps were obtained.

Figure 8. Image label example, where each pixel is classified as a class and its corresponding depth channel takes value 1. Adapted Digital Image. Jordan, J. *An overview of semantic image segmentation.* (2018). https://www.jeremyjordan.me/semantic-segmentation.



This new automated way to create labels solved the inefficiencies of the previous method. Labeling each image individually was no longer needed, saving a substantial amount of time. Additionally, the knowledge of the threshold values of each class allowed correctly classifying every pixel, thereby obtaining an exact label.

In order to visualize the label, the array needed to be reshaped into (number of images, height, width, 3(RGB)). First, the position of the maximum value on the fourth dimension (i.e. the integer assigned to the class) was taken. This showed the regions of the image where each

class was present (see Figure 9). Lastly, a different color multiplier (a three-dimensional RGB array) was applied to each class.

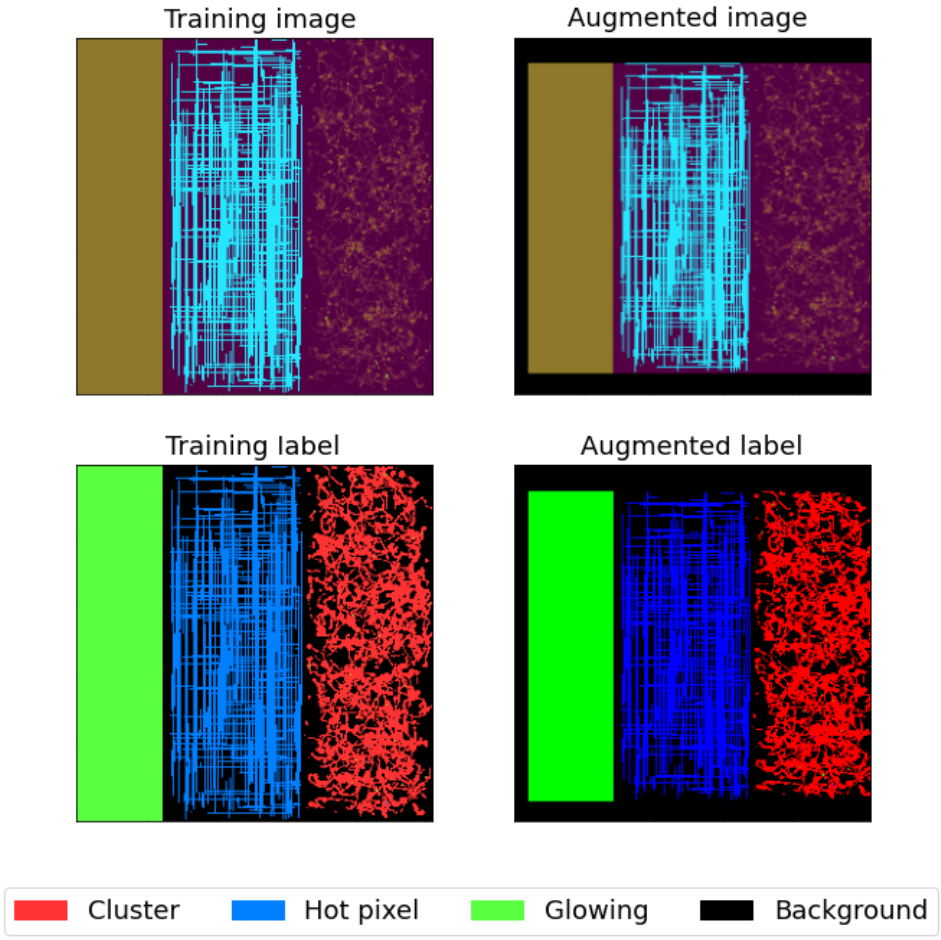
Figure 9. Segmentation map example. Adapted Digital Image. Jordan, J. *An overview of semantic image segmentation.* (2018). https://www.jeremyjordan.me/semantic-segmentation.

3.3 Data augmentation

Data augmentation is especially useful to teach invariance properties to the network. Augmentation is applied when only a few training samples are available, or when the desired property is not present in the dataset.

The lack of samples did not pose a problem, since the images were simulated. However, when the first labeling method was being employed, augmentation was applied in order to shorten the time-consuming labeling process. Labels were augmented together with images with the purpose of having a larger dataset. These augmentation transformations included rotations, translations, scaling and cropping (see Figure 10).

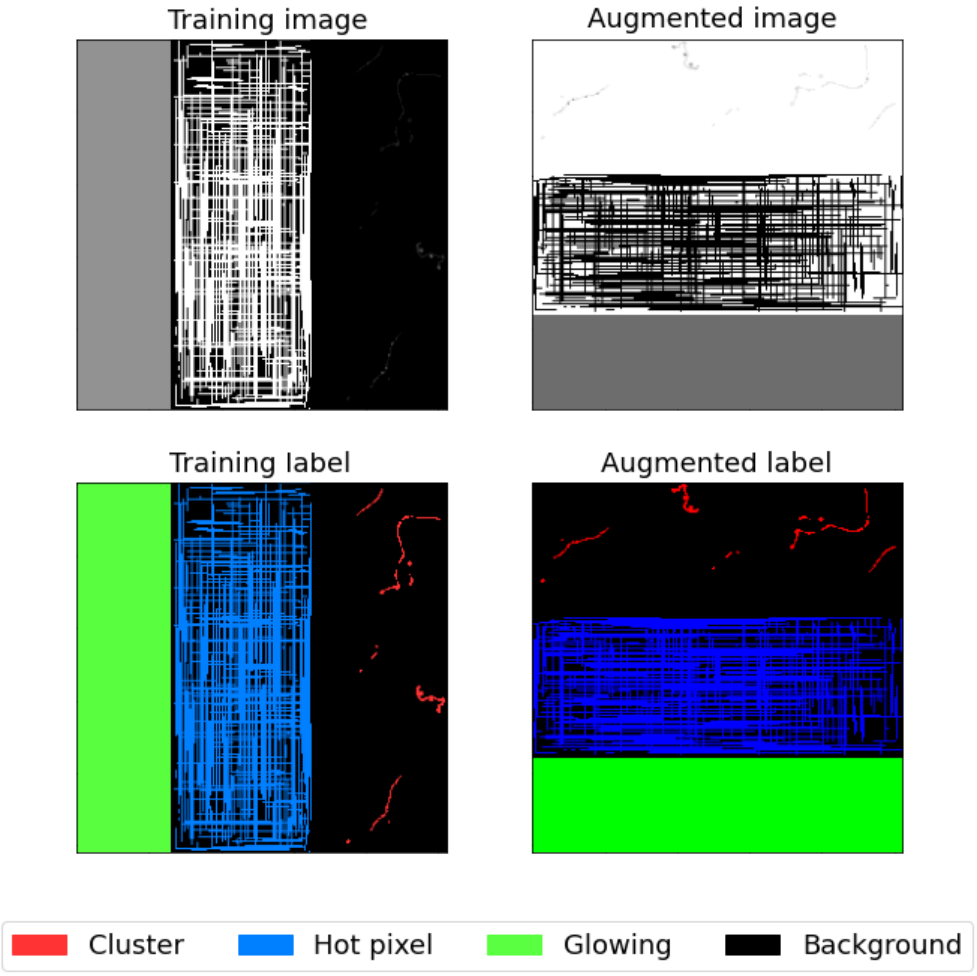
Figure 10. Image and label augmentation example. The applied transformations are translation and scaling.



On the other hand, when the automated labeling method was applied, the number of samples and labels was no longer a concern. Nevertheless, possessing data with certain properties that could not be simulated was essential in the project. A color change invariant model could be trained owing to augmentation; the images were transformed into rotated grayscale images, and color channel inversion[7] was performed (see Figure 11). This was key in the accomplishment of the final model.

[7]Color channel inversion inverts all pixel values in an image, e.g. in the standard RGB value range of [0,255], turns 5 into 255-5=250.

Figure 11. Image and label augmentation example on a grayscale image. The applied transformations are rotation and color channel inversion.

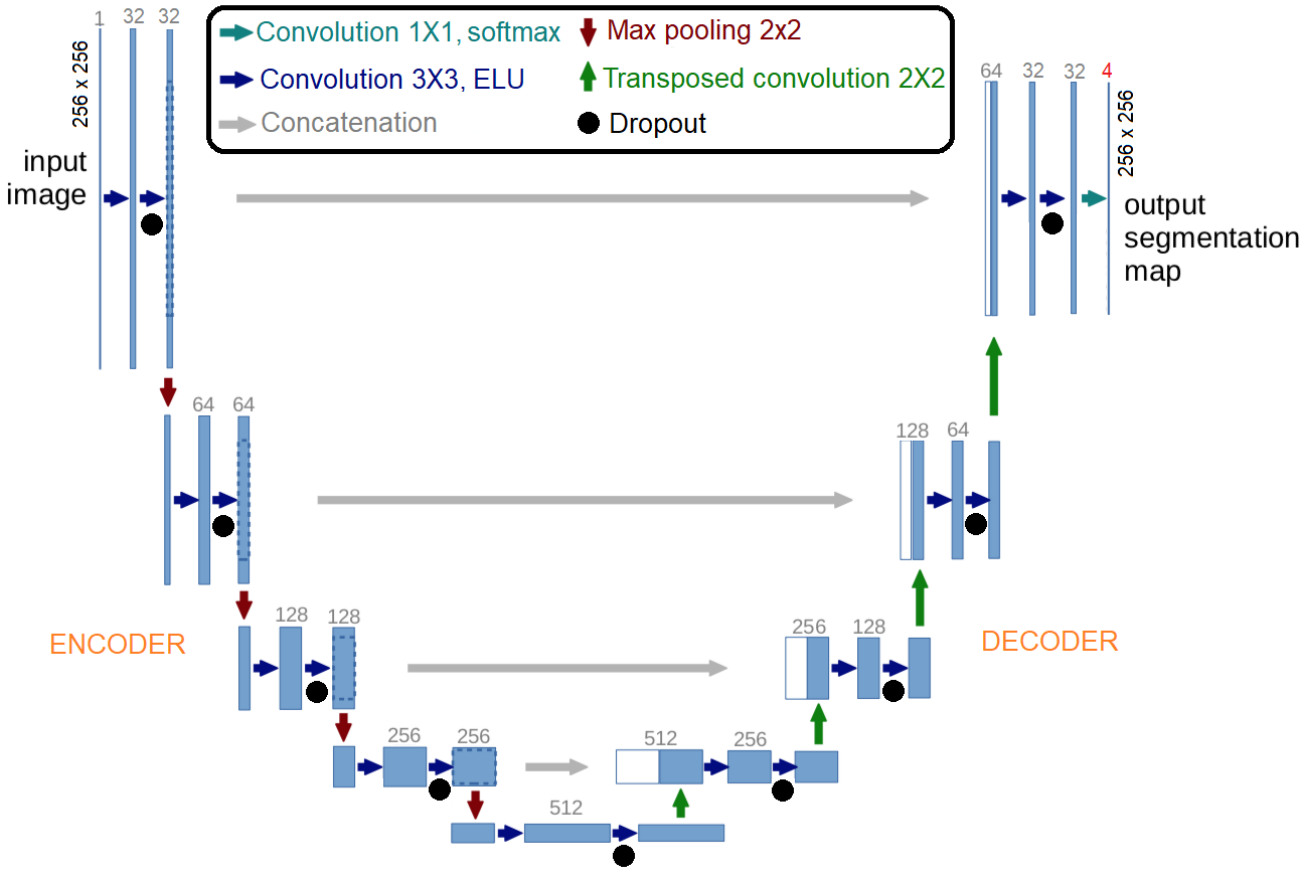


3.4 Network implementation

Due to its capacity to work efficiently with a reduced amount of images and to detect small size objects, a U-Net structure was implemented (see Figure 12). The structure, originally developed by O. Ronneberger et al.[8], is constituted by two main parts: the encoder and the decoder.

[8]Ronneberger, O. et al. *Convolutional Networks for Biomedical Image Segmentation.* (2015). https://arxiv.org/pdf/1505.04597.pdf.

Figure 12. Implemented U-Net architecture. Adapted from Ronneberger, O. et al. *Convolutional Networks for Biomedical Image Segmentation.* (2015). https://arxiv.org/pdf/1505.04597.pdf.



The encoder, following a typical CNN architecture, consisted of the repeated application of two 3×3[9] convolutional layers along with a 2×2 max pooling layer. The decoder, on the other side, consisted of an iteration of a 2×2 transposed convolution, a concatenation with the down sampling layer at the corresponding level, and two 3×3 convolutional layers. The last layer was a 1×1 convolutional layer with a softmax activation function, which returned a four-dimensional vector with the output of the previous layer transformed into a probability distribution ranged between 0 and 1.

[9]Referred to the filter size.

The total number of convolutional layers in this project was 19, in contrast to the 23 in the original architecture. The model was not able to properly train with the initial structure, due to the excessive depth; the images were being overly down sampled and the model was struggling to learn such small details. For this reason, one level of layers was removed from the structure, leading to a proper model training.

The implemented network also differentiated in the activation function (except in the last convolutional layer); the ELU (Exponential Linear Unit) was applied, instead of the ReLU (Rectified Linear Unit) function. The ELU function was found to be slower to train, but produced more accurate results.

The original loss function, on the other side, was the categorical crossentropy. This function is used on multi-class classification applications, where the last activation function outputs a probability distribution vector. However, its employment did not allow obtaining meaningful predictions. This loss function gives the same importance to all classes, no matter how frequent they are in the dataset. Since the simulated set contained background pixels in its majority, this class had a significantly greater impact on the loss function. For this reason, in order to work with the imbalanced dataset, a weighted version of the loss function was implemented. With the weighted categorical crossentropy, the least frequent classes are given the highest weight, or importance, thereby balancing the impact of all classes on the loss function.

The optimizer was not originally detailed, therefore the most common ones were tested: Adam, Adagrad, Adadelta and SGD. All of them are

adapted stochastic gradient descent methods. These methods are the algorithms that change the weights of the activation function in order to reduce the loss given by the loss function. An optimizer is defined by its learning rate. This hyper-parameter determines the amount of weights that are updated at each training iteration. The larger the learning rate, the faster the optimizer will minimize the loss function. However, if the learning rate is too large, the optimizer might not be able to converge and minimize the loss function. In the end, the chosen optimizer was adadelta, a method which dynamically adapts its learning rate over time. The automatic learning rate setting was found to be highly convenient, and worked efficiently on the simulated dataset.

Finally, unlike in the original structure, dropout layers were added between every pair of convolutional layers. The intention was to avoid overfitting as much as possible.

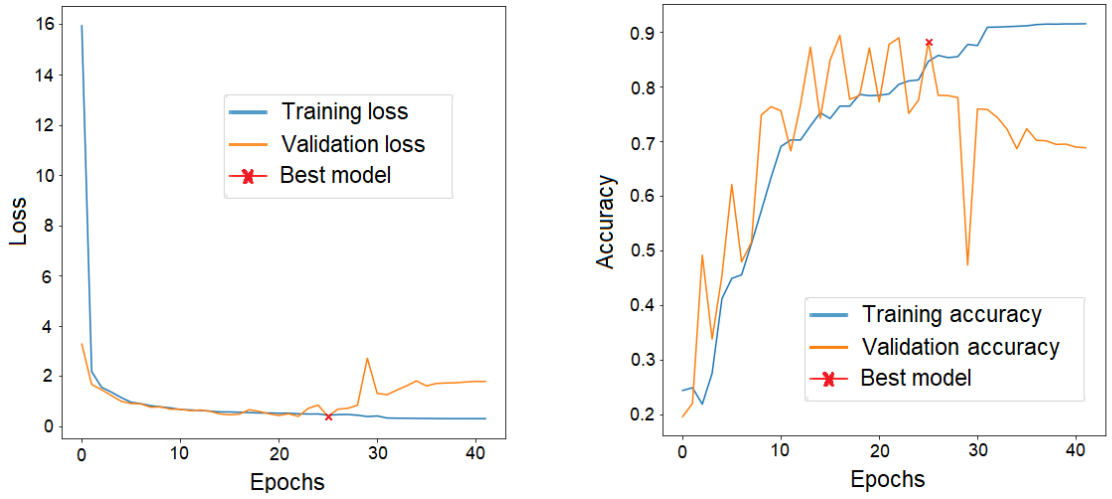
3.5 Model training

Previously, the creation of two datasets was explained: the training and testing sets. As a matter of fact, the training set was splitted into a training and a validation set. The validation set can be understood as a testing set that is applied while training. It is essential to verify that the network has not memorized the training data (i.e. overfitting) and will later be able to reliably perform on the 'unseen' test dataset.

The training dataset was shuffled and divided into batches. Then, these batches were passed through the network a certain number of times, de-

fined as epochs. A small batch size introduces a high variation within each batch, as it is improbable that a small number of training samples represent the dataset reasonably. Nonetheless, choosing a large batch size may tend to overfit the data. During the training process, the model is expected to stabilize at its optimum state, and converge its loss and accuracy (see Figure 19). However, models can occasionally be overtrained after reaching their optimum state and cause overfitting (see Figure 13). In order to avoid this, the training process was programmed to stop if the validation loss had not improved in 20 epochs, and only the model with the lowest loss was saved.

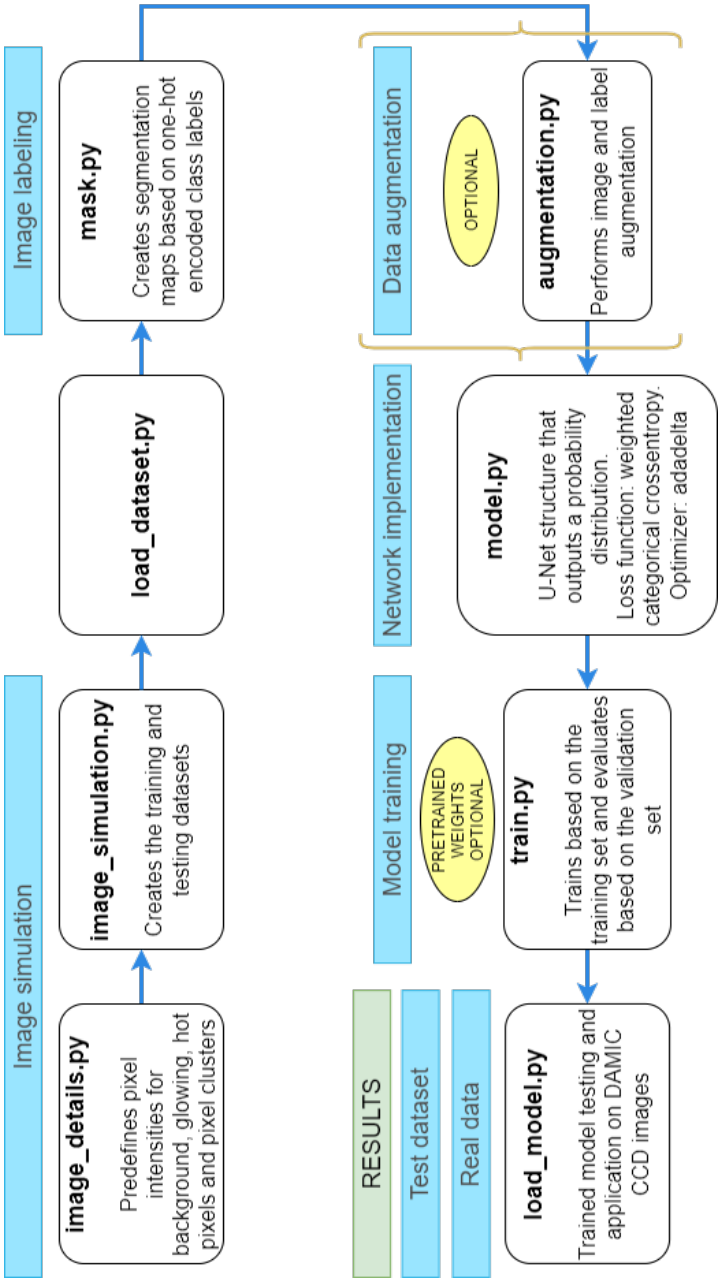
Figure 13. Example of an overtrained model that overfits after reaching its optimum state at epoch 25. *Left:* Training and validation loss. *Right:* Training and validation accuracy.



There exists no rule on which layers, activation function, loss function, optimizer or batch size work better, therefore various models had to be tested. Not only all parameters had to be tested, but even different combinations of them, in order to find out which ones worked most efficiently on the given set. Considering that a 100 epoch training took more than 5h ($\sim$ 3-4 minutes per epoch), the structure and parameter optimization of the model ended up being especially time-consuming.

The Python application consisted on 8 files, each of them with a particular implementation purpose. A summary is shown in Figure 14. See Appendix for technical specifications of the implementation.

Figure 14. Python implementation summary.



4 Results

While the implementation process already poses a challenge itself, a seemingly correct model structure and parameter choice does not always translate into a correct model training. The goal of this application was to train a suitable model for the given dataset, but most importantly, for the CCD images from DAMIC. All the challenges that had to be faced, together with the different attempts to overcome them in order to obtain the desired results, are revealed in this section.

4.1 Test dataset

For this project 200 training and 42 test images were created. As previously explained, each image contained glowing on the left side, and hot pixels and clusters randomly placed on the right (see Figure 6). From the training set 42 samples were taken for validation. The network was set with an 18% dropout. Small variations of this value (10-25% is the common dropout range) did not significantly alter the final result. Taking the original U-Net model as a reference, a batch size of 1 sample was set. The training was defined for 100 epochs.

4.1.1 Original U-Net

At the first attempt, the original U-Net architecture and parameters were employed. The trained model predicted all pixels on the test set

as background. These meaningless predictions were mainly caused by the loss function, which only focused on minimizing the loss caused by background pixels, regardless of the other three classes.

Nevertheless, this is also the reason why a misleadingly high accuracy was obtained. Due to the fact that 5% of the pixels in the dataset represented objects to detect, and all pixels were predicted as background, a 95% accuracy was achieved.

4.1.2 Hyper-parameter optimization

After the initial failure, new structures and parameters were tested in order to find the model settings that worked most efficiently on the dataset. The optimized model was trained, achieving a 98% accuracy. All background, glowing and hot pixels were correctly segmented. However, clusters were not detected (see Figure 16 *Bottom left*). The remaining 2% accuracy was equal to the percentage of cluster pixels in the dataset.

4.1.3 Employment of pretrained weights

In order to detect every class within an image, a new approach was required. With the intention of forcing the model to learn segmenting clusters, a first model was trained with a new set. These new images only contained clusters. Afterward, a second model was trained with images containing all classes. This time, the activation function weights were no longer randomly initialized; the trained weights from the first model

were used as pretrained weights on the second.

Despite the usage of pretrained weights, the model predicted the same as if the weights were randomly initialized; background, glowing and hot pixels were correctly segmented, while clusters were not.

4.1.4 Loss function adjustment

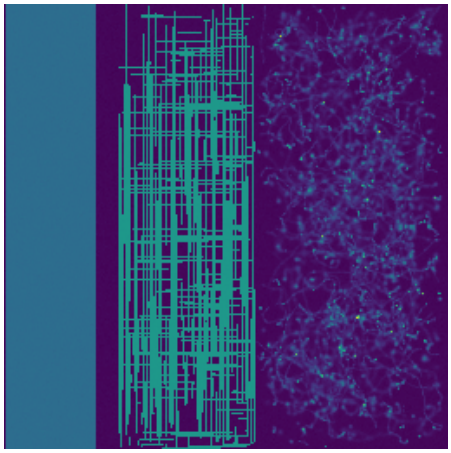
A second try to detect clusters was based on the adjustment of the loss function. Suggesting that the function was not focusing on correctly predicting clusters, the weight of this class was increased. In this occasion, glowing, hot pixels and clusters were segmented, while the background was not (see Figure 16 *Bottom middle*). The loss function was found to be significantly sensitive to its weights. Increasing the weight corresponding to clusters did translate into their detection, but at the cost of incorrectly predicting the background.

4.1.5 Creation of a balanced dataset

The previous failed attempts suggested that the issue was related to the images, rather than to the model structure and parameters. The goal was to work with more balanced loss function weights. In order to achieve that, the training set had to be modified. Greatly increasing the number of objects within the images, the percentages of each class on the dataset, and therefore the loss function weights, were more similar to each other. The new images were composed of three sections. The left side remained

unchanged, were a glowing vertical column was added. The right side was divided into two sections: one section containing hot pixels, and the other containing clusters (see Figure 15). The new dataset was constituted by 41% background, 25% glowing, 21% hot pixel and 13% cluster pixels.

Figure 15. Simulated 256×256 pixel image with a similar number of pixels belonging to each class.

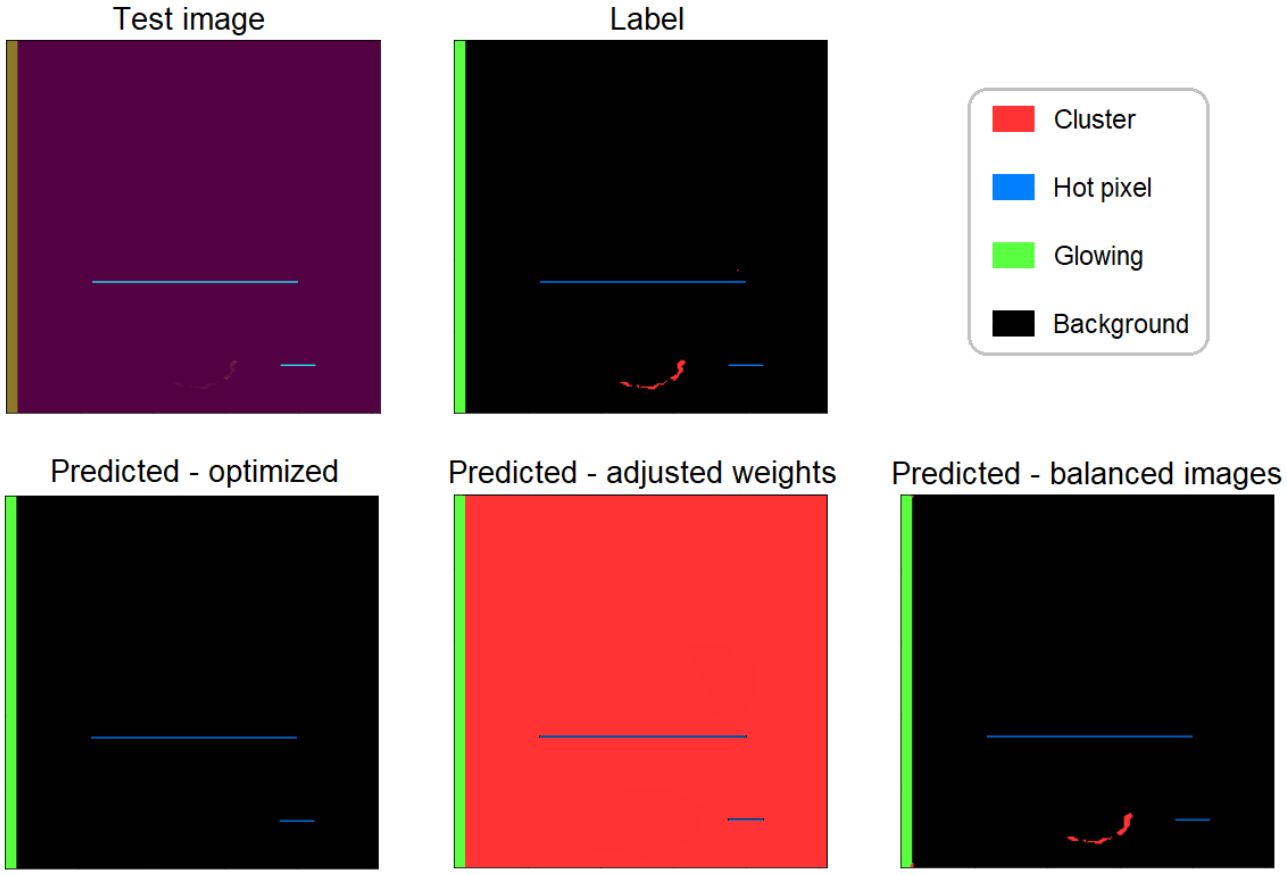


A new model was trained with the balanced dataset, and above 99% prediction accuracy was achieved in the test set, correctly segmenting every object (see Figure 16 *Bottom right*).

4.2 DAMIC data

DAMIC data was received in FITS (Flexible Image Transport System) file format. These files were read and saved as arrays that contained the collected charge by each CCD pixel. Since 256×256 pixel images were used for training the model, the DAMIC image was divided into sections of the same size. Next, each section was passed trough the trained model, obtaining the respective predicted labels. In the end, all predictions were reconstructed into a full segmentation map.

Figure 16. *Top left:* test image to be passed through the trained model. *Top middle:* correct label of the test image. *Top right:* classes to segment in a label. *Bottom left:* predicted label by the model trained after the hyper-parameter optimization. *Bottom middle:* predicted label by the model trained after the loss function adjustment. *Bottom right:* predicted label by the model trained with the balanced dataset.



4.2.1 First test

A DAMIC image was tested by the model trained with the balanced dataset. Although this model had correctly worked on the test set, an undoubtedly incorrect output was obtained for the real image; all pixels were predicted as clusters. This result forced the need to take a step back and analyze the simulated images once again.

4.2.2 Color change invariance testing

Reflecting on the reason of the failure, the color of the images was proposed as a possible cause. It seemed like the model was learning to segment objects based on color. In order to confirm the hypothesis, two tests were made.

On the first test, the color of some samples was changed. The modified images were passed through the trained model, which failed to correctly label the objects (see Figure 17 *Bottom second*).

The second test consisted on a binary classification model. It was trained to segment two classes: background and object, without distinction of the type. Yet again, when a modified image was passed through the model, an incorrect output was obtained (see Figure 17 *Bottom third*).

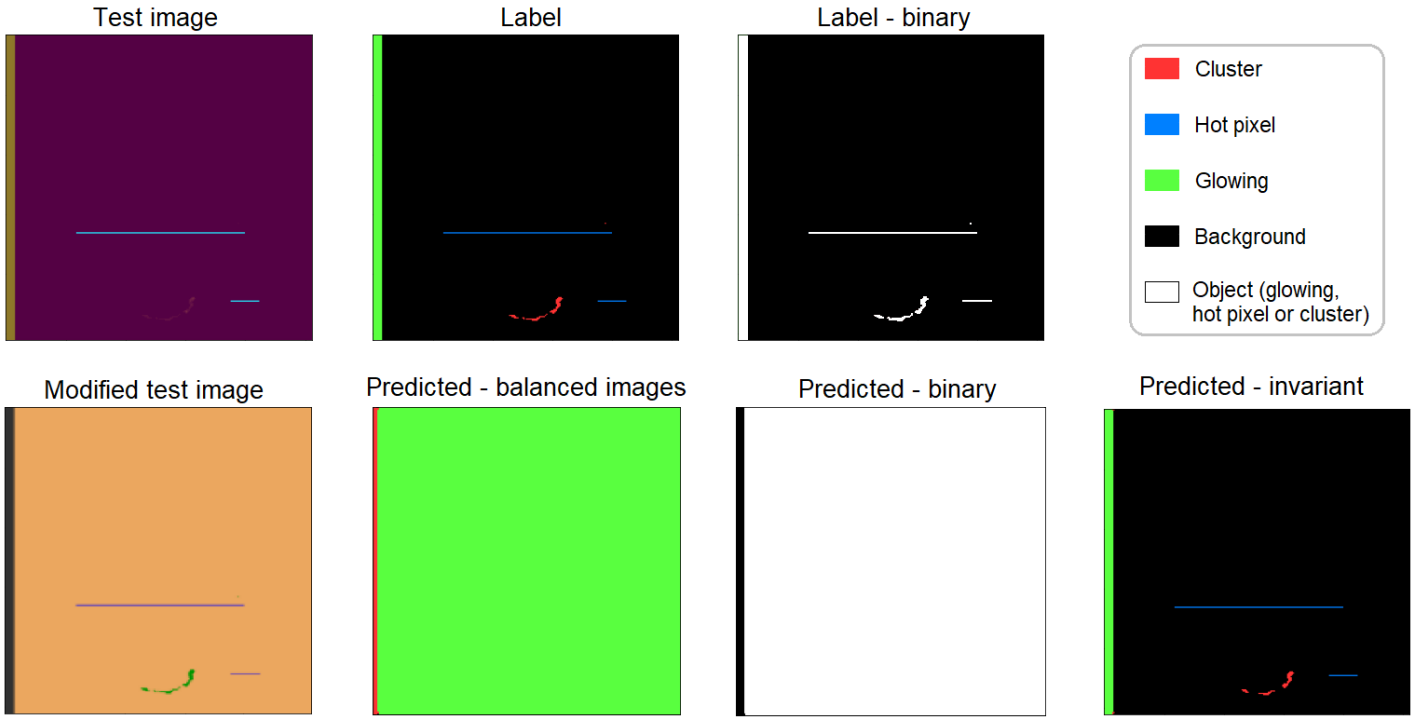
4.2.3 Color change invariance learning

At this stage begun the attempt to teach color change invariance to a model. In order to achieve this goal, two methods were studied.

First, data augmentation was applied to randomly change the color of images. A model was trained with the modified images, and consequently tested. The model correctly predicted the label of the non-modified samples. When the modified samples were tested, on the other hand, the model failed, as it did when no data augmentation was applied.

As a last attempt, data augmentation was applied to generate grayscale images, rotate them, and invert their pixel intensity values (see Figure 11). The intention was to prevent the model from learning which class should have a higher pixel intensity. Moreover, the predictions should not depend on the orientation of the objects. A new model was trained and tested. The model was able to correctly segment and label every object in non-modified and modified samples (see Figure 17 *Bottom fourth*). Nevertheless, when a DAMIC image was tested, all pixels were still being classified as clusters.

Figure 17. *From left to right. Top first:* test image to be passed through the trained model. *Top second:* correct label of the test image. *Top third:* correct binary classification label of the test image. *Top fourth:* classes to segment in a label. *Bottom first:* test image after a random color change. *Bottom second:* predicted label by the model trained with the balanced dataset. *Bottom third:* predicted label by the binary classification model. *Bottom fourth:* predicted label by the color change invariant model.



4.2.4 Image normalization

The simulated datasets were being loaded as PNG images, meaning that standard RGB pixel values were being used, which range between 0 and 255. These were hundreds of times smaller than the pixel intensity values contained in real data. For that reason, a DAMIC image normalization was proposed. However, it was proved to be ineffective; when real data was normalized to 255, the model predicted all pixels as the same class.

4.2.5 Dataset loading method modification

Trying to find a new solution, the way in which the simulated datasets were loaded was changed. Instead of loading them as PNG files, the arrays containing the predetermined pixel intensities were loaded. With this new data loading method a model was trained. When a DAMIC image was tested, an unusual signal was found; vertical lines of glowing were being predicted, coinciding with the beginning of each 256×256 section (see Figure 18).

Therefore, a last modification was made to the simulated sets; only 60% of the images contained glowing, and it did not always start from the first pixel. This way, the model did not learn that all predictions should have a glowing column, nor where should it be. The loss (see Figure 19 *Left*) and accuracy (see Figure 19 *Right*) of the training and evaluation set verified that the model did not suffer from overfitting.

Figure 18. *Left:* A 4096×4096 pixel DAMIC image (T=140K). The pixel intensity values are given in ADCs. *Right:* Predicted label with unusual vertical lines of glowing every 256 pixels.

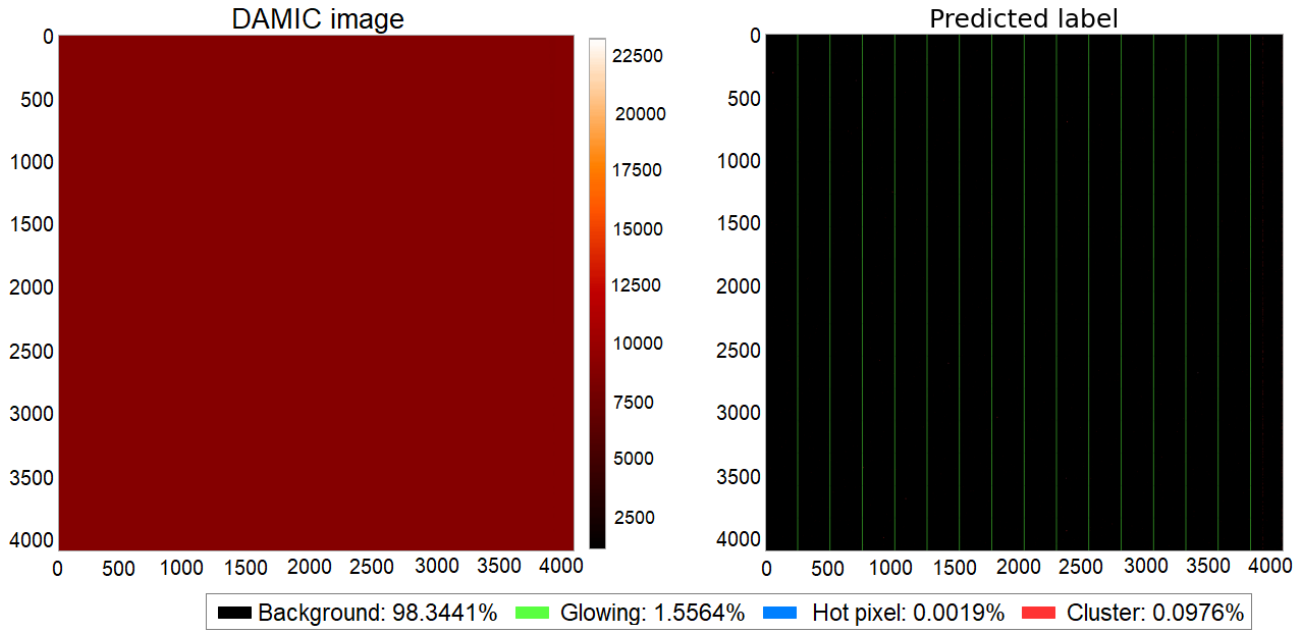
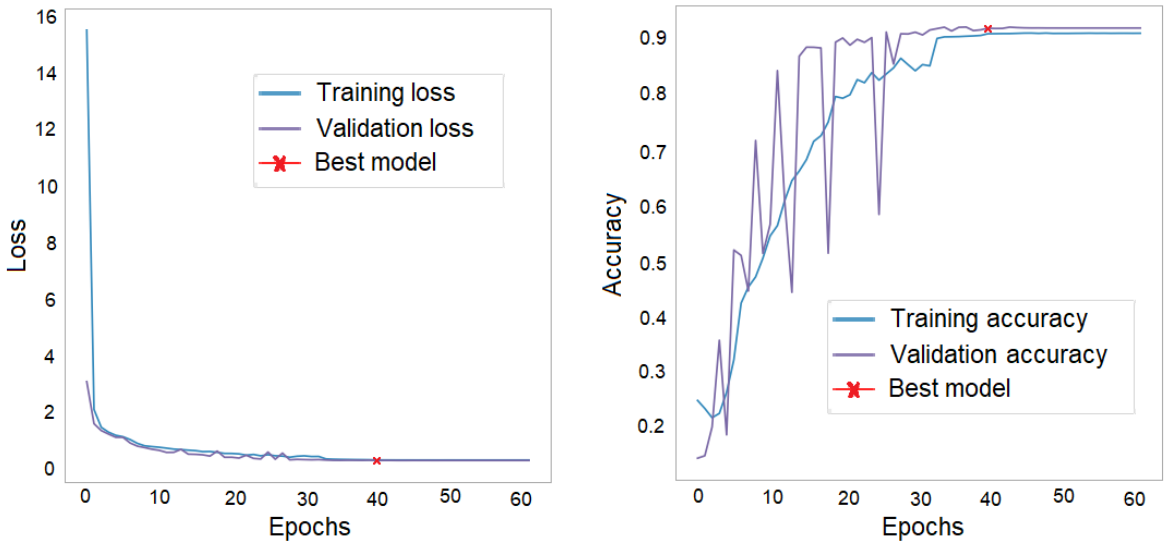


Figure 19. Learning process of the last model, trained with arrays containing the predetermined pixel intensities. *Left:* Training and validation loss. *Right:* Training and validation accuracy.



The model performed correctly on the test dataset, segmenting every object and reaching a 99.2% accuracy. The classification report showed how efficiently each class performed (see Table 1).

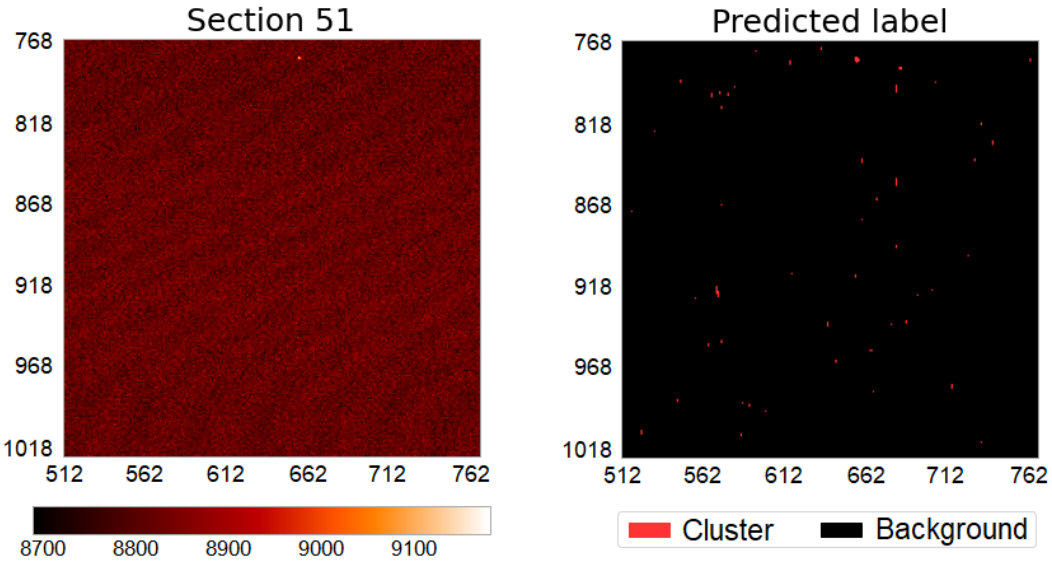
Table 1. Classification report of the predictions on the test set.

Class	*Precision*	*Recall*	*F1-score*	*Support*
Background	1.00	0.99	1.00	2535931
Glowing	1.00	1.00	1.00	187442
Hot pixel	0.99	0.98	0.95	10594
Cluster	0.44	0.63	0.52	18545

Precision: percentage of correctly classified pixels among all pixels classified as the class.
Recall: percentage of correctly classified pixels among all pixels that truly are of the class.
F1-score: harmonic mean between precision and recall.
Support: number of pixels of the class in the dataset.

The model also gave a seemingly correct prediction of a DAMIC image (T=140K). Due to the small size of the objects, these could not be seen when the whole image was displayed. If the 256×256 sections were individually observed instead, the segmented clusters could be analyzed (see Figure 20).

Figure 20. *Left:* A 256×256 pixel DAMIC image (T=140K) section. The pixel intensity values are given in ADCs. *Right:* Predicted label with segmented clusters.



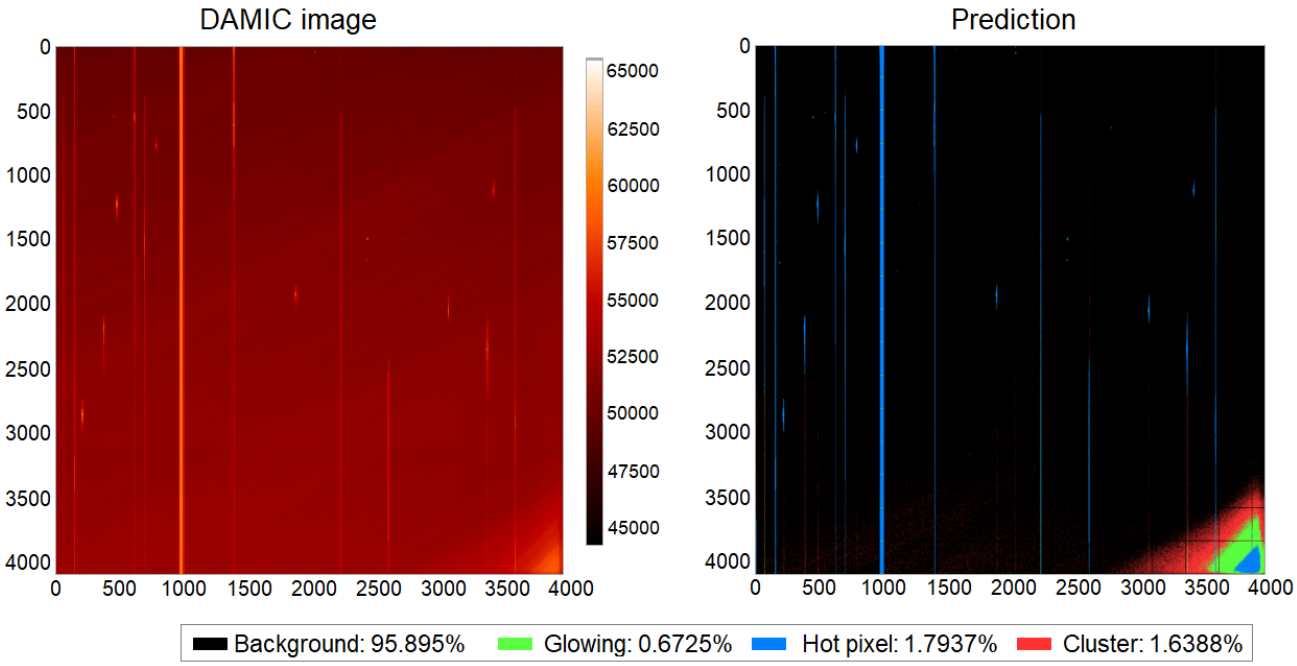
5 Discussion and future work

The relevance of the network structure and parameters has been evaluated, as shown by the inaccurate output given by the original U-Net settings. Furthermore, the results contribute a clearer understanding of how the model learns from the dataset. In spite of the employment of the weighted categorical crossentropy, the model is not able to segment all categories when it is being trained with an imbalanced set. The influence of data on the performance of the model is also proved by the unsuccessful predictions on DAMIC images. Figure 18 provides an insight into how the model learns from data. These findings should be taken into account when considering to improve the model.

The study shows the good performance of the U-Net structure detecting small size objects and its capability to work with less images than other networks. By its implementation, an automatic data quality monitoring system is created, supporting the potential of deep learning techniques in DM searches.

Nonetheless, the application's shortcomings are revealed when a more complex image is passed through the model, obtaining a prediction that is not entirely correct (see Figure 21). In order to deal with this issue, defining geometrical restrictions is a possible solution. For instance, imposing a hot pixel prediction to have its expected shape, which is a vertical or horizontal line. Likewise a maximum cluster size can be defined. Additionally, glowing predictions are likely to improve if different, more realistic shapes are generated in the simulated dataset.

Figure 21. *Left:* DAMIC image (T=240K). The pixel intensity values are given in ADCs. *Right:* Predicted label with unusual hot pixel and cluster segmentation on the bottom right side.



The results might suggest that a model is limited to the range of intensities to which it is trained. However, applying different normalization techniques, such as linear scaling[10] or clipping[11], to the DAMIC images is a seemingly good approach to solve this limitation.

Moreover, in future work more classification categories can be added, allowing particle identification by measuring their energy, and possible DM signals could be discerned.

[10]Linear scaling converts the values from their natural range into a standard range, usually [0,1].
[11]Clipping caps all values above or below a certain value to a fixed value.

Appendix A

The source code of the application was implemented in Python, with the requirement of the following open-source libraries (version numbers are up to date: 21.06.2020):

- Keras (2.3.1)
- TensorFlow (2.1.0)
- NumPy (1.18.1)
- scikit-learn (0.22.1)
- scikit-image (0.9.3)
- OpenCV-Python (4.0.1)
- imgaug (0.4.0)
- Matplotlib (3.2.1)
- Pillow (7.1.2)

The core files of the implementation are shown in Appendix A.1 and Appendix A.2. All the files can be found on GitHub: https://github.com/aritzLizoain/Image-segmentation

Appendix A.1

The U-Net structure is implemented in the model.py file. Every layer composing the CNN and each hyper-parameter is specified in it. In addition, the weighted categorical crossentropy loss function is defined.

```
"""
@author: Aritz Lizoain
ARCHITECTURE: U-Net
Input: images (n_img, h, w, 3(rgb)) and labels (n_img, h, w, 4)
Output: predicted label (n_img, h, w, 4)
"""

import numpy as np
import os
import matplotlib.pyplot as plt
from keras.models import *
from keras.layers import *
from keras.optimizers import *
from keras.callbacks import ModelCheckpoin
from keras.callbacks import LearningRateScheduler
from keras import backend as keras
import keras.losses
from keras import regularizers
import keras.backend as K

#####################################################################
#WEIGHTED CROSSENTRPY  LOSS FUNCTION (For imbalanced datasets)
def weighted_categorical_crossentropy(weights= [1.,1.,1.,1.]):
    print('Loss function: weighted categorical crossentropy')
    def wcce(y_true, y_pred):
        Kweights = K.constant(weights)
        if not K.is_tensor(y_pred): y_pred = K.constant(y_pred)
        y_true = K.cast(y_true, y_pred.dtype)
        return K.categorical_crossentropy(y_true, y_pred) *\
            K.sum(y_true * Kweights, axis=-1)
    return wcce

#-------------------------------------------------------------------
def unet(pretrained_weights = None, input_size = (256,256,1),\
         weights = get_weights(train_images,test_images),\
         activation = 'elu', dropout = 0.18, loss=\
         weighted_categorical_crossentropy(weights), optimizer\
             ='adadelta', reg = 0.01):

    inputs = Input(input_size)
    s = Lambda(lambda x: x / 255) (inputs)

    """CONTRACTIVE Path (ENCODER)"""

    c1 = Conv2D(32, 3 , activation, kernel_initializer=\
                'he_normal', padding='same',\
                kernel_regularizer=regularizers.l2(reg)) (s)
    c1 = Dropout(dropout) (c1)
```

```
49      c1 = Conv2D(32, 3 , activation, kernel_initializer=\
50                  'he_normal', padding='same',\
51                  kernel_regularizer=regularizers.l2(reg)) (c1)
52      p1 = MaxPooling2D((2, 2)) (c1)
53
54      c2 = Conv2D(64, 3, activation, kernel_initializer=\
55                  'he_normal', padding='same',\
56                  kernel_regularizer=regularizers.l2(reg)) (p1)
57      c2 = Dropout(dropout) (c2)
58      c2 = Conv2D(64, 3, activation, kernel_initializer=\
59                  'he_normal', padding='same',\
60                  kernel_regularizer=regularizers.l2(reg)) (c2)
61      p2 = MaxPooling2D((2, 2)) (c2)
62
63      c3 = Conv2D(128, 3, activation, kernel_initializer=\
64                  'he_normal', padding='same',\
65                  kernel_regularizer=regularizers.l2(reg)) (p2)
66      c3 = Dropout(dropout) (c3)
67      c3 = Conv2D(128, 3, activation, kernel_initializer=\
68                  'he_normal', padding='same',\
69                  kernel_regularizer=regularizers.l2(reg)) (c3)
70      p3 = MaxPooling2D((2, 2)) (c3)
71
72      c4 = Conv2D(256, 3, activation, kernel_initializer=\
73                  'he_normal', padding='same',\
74                  kernel_regularizer=regularizers.l2(reg)) (p3)
75      c4 = Dropout(dropout) (c4)
76      c4 = Conv2D(256, 3, activation, kernel_initializer=\
77                  'he_normal', padding='same',\
78                  kernel_regularizer=regularizers.l2(reg)) (c4)
79      p4 = MaxPooling2D(pool_size=(2, 2)) (c4)
80
81      c5 = Conv2D(512, 3, activation, kernel_initializer=\
82                  'he_normal', padding='same',\
83                  kernel_regularizer=regularizers.l2(reg)) (p4)
84      c5 = Dropout(dropout) (c5)
85      c5 = Conv2D(512, 3, activation, kernel_initializer=\
86                  'he_normal', padding='same',\
87                  kernel_regularizer=regularizers.l2(reg)) (c5)
88
89      """EXPANSIVE Path (DECODER)"""
90
91      u6 = Conv2DTranspose(256, 2, strides=(2, 2),\
92                           padding='same') (c5)
93      u6 = concatenate([u6, c4])
94      c6 = Conv2D(256, 3, activation, kernel_initializer=\
95                  'he_normal', padding='same',\
96                  kernel_regularizer=regularizers.l2(reg)) (u6)
```

```
97      c6 = Dropout(dropout) (c6)
98      c6 = Conv2D(256, 3, activation, kernel_initializer=\
99                  'he_normal', padding='same',\
100                 kernel_regularizer=regularizers.l2(reg)) (c6)
101
102     u7 = Conv2DTranspose(128, 2, strides=(2, 2),\
103                          padding='same') (c6)
104     u7 = concatenate([u7, c3])
105     c7 = Conv2D(128, 3, activation, kernel_initializer=\
106                 'he_normal', padding='same',\
107                 kernel_regularizer=regularizers.l2(reg)) (u7)
108     c7 = Dropout(dropout) (c7)
109     c7 = Conv2D(128, 3, activation, kernel_initializer=\
110                 'he_normal', padding='same',\
111                 kernel_regularizer=regularizers.l2(reg)) (c7)
112
113     u8 = Conv2DTranspose(64, 2, strides=(2, 2),\
114                          padding='same') (c7)
115     u8 = concatenate([u8, c2])
116     c8 = Conv2D(64, 3, activation, kernel_initializer=\
117                 'he_normal', padding='same',\
118                 kernel_regularizer=regularizers.l2(reg)) (u8)
119     c8 = Dropout(dropout) (c8)
120     c8 = Conv2D(64, 3, activation, kernel_initializer=\
121                 'he_normal', padding='same',\
122                 kernel_regularizer=regularizers.l2(reg)) (c8)
123
124     u9 = Conv2DTranspose(32, 2, strides=(2, 2),\
125                          padding='same') (c8)
126     u9 = concatenate([u9, c1], axis=3)
127     c9 = Conv2D(32, 3, activation, kernel_initializer=\
128                 'he_normal', padding='same',\
129                 kernel_regularizer=regularizers.l2(reg)) (u9)
130     c9 = Dropout(dropout) (c9)
131     c9 = Conv2D(32, 3, activation, kernel_initializer=\
132                 'he_normal', padding='same',\
133                 kernel_regularizer=regularizers.l2(reg)) (c9)
134
135     #softmax activaition function of the last layer
136     outputs = Conv2D(4, 1, activation='softmax') (c9)
137
138     model = Model(inputs=[inputs], outputs=[outputs])
139     model.compile(optimizer=optimizer, loss=loss,\
140                   metrics = ['accuracy'])
141     if(pretrained_weights):
142         print('Pretrained weights'.format(pretrained_weights))
143         model.load_weights(pretrained_weights)
144     return model
```

Appendix A.2

Train.py and load_model.py are the files that are executed. Every function defined in the rest of the files is imported by them. Here, a combination of both files is shown. First, the data is loaded, and the labels are created. Then, data augmentation is applied. Next, the model is trained with the determined hyper-parameters. After that, the model is evaluated and tested; the accuracy and loss are plotted, the test set is passed through the model, and the classification report is displayed. Last, the DAMIC image prediction is obtained.

```
"""
@author: Aritz Lizoain
TRAINING + TESTING + DAMIC IMAGE PREDICTION
Working directory must be where all files are located.
"""

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
DATA LOADING, LABEL CREATION AND DATA AUGMENTATION
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
import numpy as np
import random
import matplotlib.pyplot as plt
from mask import *
from load_dataset import load_images
from augmentation import *

#############################################################
"""Load dataset ARRAYS (predefined pixel intensities)"""
images_original = np.load('Images/Train/training_data.npy')
test_images_original = np.load('Images/Test/test_data.npy')
print('Data arrays correctly loaded')

"""Create labels"""
#create_masks() from mask.py
print('Creating training image labels...')
masks = create_masks(images_original)
print('Creating test image labels...')
test_masks = create_masks(test_images_original)
```

```
29
30 """Create labels"""
31 #create_labels() from mask.py. For label visualization.
32 print('Creating training image labels...')
33 labels = create_labels(images_original)
34 print('Creating test image labels...')
35 test_labels=create_labels_noStat_noPrint(test_images_original)
36
37 """Augmentation"""
38 #augmentation_operationName() from augmentation.py
39 #Augmented images and labels are added to the datasets
40 images_augmented, labels_augmented = augmentation_Invert\
41     (images_original, labels)
42
43 """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
44 MODEL TRAINING
45 """"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
46 from models import unet, weighted_categorical_crossentropy
47 from keras.callbacks import EarlyStopping, ModelCheckpoint
48 from keras.callbacks import ReduceLROnPlateau, CSVLogger
49 from load_dataset import get_weights
50
51 ##################################################################
52 """HYPERPARAMETERS"""
53 #-Model-----------------------------------------------------------
54 split=0.21 # Validation and training dataset split
55 pretrained_weights = None
56 input_size = (IMG_HEIGHT, IMG_WIDTH, 1)
57 weights = get_weights(images_original,test_images_original)
58 activation = 'elu'
59 dropout = 0.18
60 loss = weighted_categorical_crossentropy(weights)
61 optimizer = 'adadelta'
62
63 #-Fit-------------------------------------------------------------
64 epochs = 100
65 batch_size = 1
66 callbacks=[EarlyStopping(patience=20,verbose=1),\
67            ReduceLROnPlateau(factor=0.1, patience=5,\
68                              min_delta=0.001,min_lr=0.0000001,\
69                               verbose=1),ModelCheckpoint\
70                ('Models/modelName.h5'.format(epochs),\
71                 save_best_only=True,save_weights_only=False),\
72                     CSVLogger('Models/modelName.log')]
73 #module 'h5py' has no attribute 'Group' <--folder doesn't exist
74
75 #-----------------------------------------------------------------
76 """Structure settings"""
```

```
model = unet(pretrained_weights, input_size, weights,\
         activation, dropout, loss, optimizer)
#one more dimension needs to be created in order to train
images_all=images_original[..., np.newaxis]
test_images_all = test_images_original[..., np.newaxis]
#normalize
# normalization_value = 255
# images_all = images_all/images_all.max()*normalization_value

"""Training"""
results = model.fit(images_all, masks, validation_split,\
                    epochs,batch_size,callbacks, shuffle=True)
print('Model correctly trained and saved')

"""Loss plot"""
plt.figure(figsize=(8, 8))
plt.grid(False)
plt.title("Learning curve LOSS", fontsize=25)
plt.plot(results.history["loss"], label="Loss")
plt.plot(results.history["val_loss"], label="Validation loss")
p=np.argmin(results.history["val_loss"])
plt.plot( p, results.history["val_loss"][p], marker="x",\
         color="r", label="best model")
plt.xlabel("Epochs", fontsize=16)
plt.ylabel("Loss", fontsize=16)
plt.legend();
plt.savefig(TEST_PREDICTIONS_PATH+'Loss')

"""Accuracy plot"""
plt.figure(figsize=(8, 8))
plt.grid(False)
plt.title("Learning curve ACCURACY", fontsize=25)
plt.plot(results.history["accuracy"], label="Accuracy")
plt.plot(results.history["val_accuracy"],\
         label="Validation Accuracy")
plt.plot( p, results.history["val_accuracy"][p], marker="x",\
         color="r", label="best model")
plt.xlabel("Epochs", fontsize=16)
plt.ylabel("Accuracy", fontsize=16)
plt.legend();
plt.savefig(TEST_PREDICTIONS_PATH+'Accuracy')

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
TESTING AND EVALUATING THE MODEL
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
from mask import output_to_label, get_max_in_mask
import matplotlib.patches as mpatches

```

```
125 ################################################################
126 """Model predictions"""
127 print('Testing on {0} images'.format(len(test_images_all)))
128 test_outputs = model.predict(test_images_all, verbose=1)
129 #output_to_label() from mask.py for visualization
130 print('Creating predicted labels of test images...')
131 test_outputs_labels=output_to_label(test_outputs)
132
133 #Legend
134 red_patch = mpatches.Patch(color=[1, 0.2, 0.2],\
135                            label='Cluster')
136 blue_patch = mpatches.Patch(color=[0,0.5,1.],\
137                             label='Hot pixel')
138 green_patch = mpatches.Patch(color=[0.35,1.,0.25],\
139                              label='Glowing')
140 black_patch = mpatches.Patch(color=[0./255, 0./255, 0./255],\
141                              label='Background')
142
143 """Prediction of ALL TEST samples"""
144 for ix in range (len(test_outputs)):
145     fig, ax = plt.subplots(1, 3, figsize=(20, 10))
146     ax[0].grid(False)
147     ax[0].imshow(np.squeeze(test_images_all[ix]), cmap="gray")
148     ax[0].set_title('Test image {0}'.format(ix+1), fontsize=25)
149     ax[0].set_xlabel('pixels', fontsize=16)
150     ax[0].set_ylabel('pixels', fontsize=16)
151     ax[1].grid(False)
152     ax[1].imshow(np.squeeze(test_labels[ix]))
153     ax[1].set_title('Label', fontsize=25);
154     ax[1].set_xlabel('pixels', fontsize=16)
155     ax[1].set_ylabel('pixels', fontsize=16)
156     ax[2].grid(False)
157     ax[2].imshow(test_outputs_labels[ix])
158     ax[2].set_title('Predicted label', fontsize=25);
159     ax[2].set_xlabel('pixels', fontsize=16)
160     ax[2].set_ylabel('pixels', fontsize=16)
161     plt.legend(loc='upper center', bbox_to_anchor=\
162                (-0.12, -0.15), fontsize=18,\
163                 handles=[red_patch, blue_patch, green_patch,\
164                          black_patch], ncol=4)
165     plt.savefig(TEST_PREDICTIONS_PATH+'Test{0}'.format(ix+1))
166     plt.show()
167
168 """Model evaluation"""
169 evaluation = model.evaluate(test_images_all,\
170                             test_masks, batch_size=16)
171 print('The accuracy of the model on the test set is: ',\
172       evaluation[1]*100,'%')
```

```
print('The loss of the model on the test set is: ',\
      evaluation[0])

"""Classification report"""
number_to_class = ['background', 'glowing',\
                   'hot pixel', 'cluster']
test_max_masks=get_max_in_mask(test_masks)
test_max_outputs=get_max_in_mask(test_outputs)
test_masks_array=test_max_masks.ravel()
test_outputs_array=test_max_outputs.ravel()
from sklearn.metrics import classification_report
print(classification_report(y_true = test_masks_array,\
                           y_pred = test_outputs_array,\
                           target_names=number_to_class))

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
DAMIC IMAGE (.FITS FILE)
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
"""Loading the DAMIC image and creating the sections"""
print('Loading real test image from fits file...')
size=256 #section size
normalized='no'
normalization_value = 255
name = 'DAMIC'
#process_fits() from load_dataset
image_data_use, test_images_real, details = process_fits\
    (name='Fits_files/{0}.fits'.format(name), size,\
     normalized, normalization_value)
print('> Test image {0}'.format(name))
test_images_real=test_images_real[..., np.newaxis]

"""Prediction of all the sections"""
print('  Testing on {0} real image sections'.format\
      (len(test_images_real)))
test_outputs_real = model.predict(test_images_real, verbose=1)

"""Reconstruction of the predicted labels"""
# images_small2big() from load_dataset
test_outputs_real_big = images_small2big\
    (images=test_outputs_real, details=details)
test_outputs_real_big=test_outputs_real_big[np.newaxis, ...]
unique_elements_real, percentages_real = percentage_result\
    (test_outputs_real_big)

#Legend
real_percentages = np.zeros(4)
for i in range (0, len(percentages_real)):
    real_percentages[int(unique_elements_real[i])] = \
```

```
        percentages_real[i]
Background_percentage = mpatches.Patch\
    (color=[0./255, 0./255, 0./255], label='Background: {0} %'\
     .format(real_percentages[0]))
Glowing_percentage = mpatches.Patch\
    (color=[0.35,1.,0.25], label='Glowing: {0} %'.\
     format(real_percentages[1]))
Hot_pixel_percentage = mpatches.Patch\
    (color=[0,0.5,1.], label='Hot pixel: {0} %'.\
     format(real_percentages[2]))
Cluster_percentage = mpatches.Patch\
    (color=[1, 0.2, 0.2], label='Cluster: {0} %'.\
     format(real_percentages[3]))

"""Finding a specific class"""
# Check the ones with clusters in small sections
#check_one_object() from load_dataset
check_one_object(test_outputs_real, test_images_real,\
                 object_to_find='hot pixel', real_percentages\
                     =real_percentages, details=details)
# options: 'background', 'glowing', 'hot pixel', 'cluster'

"""Prediction of the DAMIC image"""
fig, ax = plt.subplots(1, 2, figsize=(20, 10))
ax[0].grid(False)
ax[0].imshow(image_data_use)
ax[0].set_title('DAMIC image', fontsize=25);
ax[0].set_xlabel('pixels', fontsize=16)
ax[0].set_ylabel('pixels', fontsize=16)
ax[1].grid(False)
#output_to_label() from mask.py
ax[1].imshow(output_to_label(test_outputs_real_big)[0])
ax[1].set_title('Predicted label', fontsize=25);
ax[1].set_xlabel('pixels', fontsize=16)
ax[1].set_ylabel('pixels', fontsize=16)
plt.legend(loc='upper center', bbox_to_anchor=(0.15, -0.09),\
           fontsize=16,handles=\
           [Background_percentage, Glowing_percentage,\
            Hot_pixel_percentage, Cluster_percentage], ncol=4)
plt.savefig(TEST_PREDICTIONS_PATH+'DAMIC_prediction_{0}_{1}'.\
            format(model_name, name))
plt.show()
```

References

[1] Bergström, L. *Dark Matter Evidence, Particle Physics Candidates and Detection Methods.* 479–496 (Annalen der Physik, 2012). https://arxiv.org/pdf/1205.4882.pdf.

[2] Feng, J. *Dark Matter Candidates from Particle Physics and Methods of Detection.* 495-545 (Annual Review of Astronomy and Astrophysics, 2010). https://arxiv.org/pdf/1003.0904.pdf.

[3] Aghanim, N. et al. *Planck 2018 results. VI. Cosmological parameters.* (Planck Collaboration, 2018). https://arxiv.org/pdf/1807.06209.pdf.

[4] Aguilar-Arevalo, A. et al. *Measurement of radioactive contamination in the CCD's of the DAMIC experiment.* (Journal of Physics, 2015). https://arxiv.org/pdf/1506.02562.pdf.

[5] Castelló-Mor, N., DAMIC-M Collaboration. *DAMIC-M Experiment: Thick, Silicon CCDs to search for Light Dark Matter.* (Elsevier BV, 2020). https://arxiv.org/pdf/2001.01476.pdf.

[6] Pant, A. *Introduction to Machine Learning for Beginners.* (Towards Data Science, 2019). https://towardsdatascience.com/introduction-to-machine-learning-for-beginners-eed6024fdb08.

[7] Dwivedi, D. *Machine Learning For Beginners.* (Towards Data Science, 2018). https://towardsdatascience.com/machine-learning-for-beginners-d247a9420dab.

[8] Lee, J. *Intuitive Deep Learning Part 2: CNNs for Computer Vision.* (Medium, 2019). https://medium.com/intuitive-deep-learning/intuitive-deep-learning-part-2-cnns-for-computer-vision-

24992d050a27.

[9] Deshpande, A. *A Beginner's Guide To Understanding Convolutional Neural Networks*. (2017). https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/.

[10] Saha, S. *A Comprehensive Guide to Convolutional Neural Networks*. (Towards Data Science, 2018). https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[11] Torres, J. *Convolutional Neural Networks for Beginners*. (Towards Data Science, 2018). https://towardsdatascience.com/convolutional-neural-networks-for-beginners-practical-guide-with-python-and-keras-dc688ea90dca.

[12] Lee, J. *Build your first Convolutional Neural Network to recognize images*. (Medium, 2019). https://medium.com/intuitive-deep-learning/build-your-first-convolutional-neural-network-to-recognize-images-84b9c78fe0ce.

[13] Sharma, A. *Convolutional Neural Networks in Python with Keras*. (DataCamp, 2017). https://www.datacamp.com/community/tutorials/convolutional-neural-networks-python.

[14] Ronneberger, O. et al. *Convolutional Networks for Biomedical Image Segmentation*. (2015). https://arxiv.org/pdf/1505.04597.pdf.

[15] Lamba, H. *Understanding Semantic Segmentation with UNET*. (Towards Data Science, 2019). https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47.

[16] Jordan, J. *An overview of semantic image segmentation*. (2018). https://www.jeremyjordan.me/semantic-segmentation.