technische universität
dortmund

Bachelor's Thesis

Gamma-Hadron Separation with Deep Clustering - Learning to Detect Gamma-Particles with Unsupervised Representation Learning

Max Glogau

August 2021

Gutachter:
Prof. Dr. Katharina Morik
M. Sc. Lukas Pfahler

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Künstliche Intelligenz (LS-8)
https://www-ai.cs.tu-dortmund.de

Contents

Chapter 1

Introduction

1.1 Motivation

In astrophysics, machine learning is often deployed to analyze certain events occurring in the real world. One field of research is the observation of cosmic events. The First G-APD Cherenkov Telescope(FACT) has the goal of capturing images of particle showers of gamma rays. The resulting images can then be used to derive cosmic events from the energy and angle of a particle. The problem with analyzing gamma rays is the background noise, mainly produced by cosmic rays from hadrons, which gives no conclusion on a direct origin source. This is called the gamma-hadron separation problem. The traditional approach to solve this only records data if a certain energy threshold is reached. Afterward, a complex machine learning pipeline is executed to separate gamma rays from background noise. The recent approach to tackle this problem is the use of deep learning directly on the recorded datastream [3]. The deep learning model can accurately predict gamma events, better than the traditional approach, and also uses fewer resources.

The most widely used deep learning models are convolutional neural networks, trained with manually annotated datasets. To improve the generalization of these models further, the most obvious way would be the use of larger datasets to train the neural networks. The problem with larger datasets arises in the manual work, necessary to annotate the examples. Another problem with the supervised approach is the training when labeling training data is difficult. One way to label the data is through complex simulations, which are often laborious to set up. Consequently, it would be best to omit the labeling part and let the model learn the features autonomously. This does not only save manual labor to annotate examples or to set

up a simulation but also saves researchers time to focus on more important tasks. This is where unsupervised learning is beneficial. The most promising approach to train neural networks without supervision is a combination of clustering and feature learning. The basic principle of this approach is to group features into clusters and use these clusters as supervision to the neural network.

1.2 Goal

Without pre-defined features to search for, or even without labels, we need a method to learn data representations autonomously. Learning these representations is the key to a successful classification of gamma events. Therefore, the goal of this thesis is to test the approach of combining deep neural networks with clustering on images from the FACT telescope. We will train three methods that use this approach, evaluate the obtained clusters and neural networks, and compare the results to each other. As we are specifically interested in their performance on the gamma-hadron separation problem, we compare these methods, by making use of a simulated dataset with labels, as well as a significance test using additional information about the real data. By combining traditionally supervised neural networks and unsupervised clustering methods, we also want to know how these new approaches compare to the original methods. Thus, we will compare the derived clusterings of the data to a standard clustering method and test the trained neural networks against a fully supervised one. The following Section will give a brief overview of the structure of this thesis.

1.3 Structure

We start by giving some information about the theoretical background for this thesis in Chapter 2, including neural networks and traditional clustering approaches. The next Chapter 3 covers the unsupervised clustering methods used for our later comparison. It is followed by Chapter 4, where we take a deeper look at the used datasets. After a summary of the implementation process in Chapter 5, we present the conducted experiments in Chapter 6 and close with a summary of this thesis in Chapter 7.

Chapter 2

Theoretical Background

There is no universal definition for machine learning, but there are several definitions that give a broad description of the field. Murphy [18] defines machine learning as a set of methods that can find patterns in a dataset or use these patterns to make predictions or decisions. Another much broader definition by Mitchell [17] is that "Machine Learning is the study of computer algorithms that improve automatically through experience". Therefore the computer is not explicitly programmed to solve some specific task but rather learns to solve it through an iterative process.

The main subclasses of machine learning are supervised and unsupervised learning. There is another commonly used subclass, reinforcement learning, where the algorithm interacts with the environment to learn through measures like punishment and reward. This type of machine learning algorithm is not further dealt with in this thesis and is only mentioned for completeness.

In supervised learning, the algorithm learns to map an input x_n to a label y_n. Each input x_n contains multiple features. The simplest forms of features are n-dimensional vectors with attributes describing the input. More complex cases are images, sentences, or other complex structures. The name is derived from an instructor giving the labels as supervision to the learning algorithm, but is also used when labels are generated automatically [10] [18].
A simple supervised learning algorithm consists of one layer where a linear function, mapping input data to output data, is optimized. The concept of layers is further described in Section 2.1 in the context of neural networks.

In a supervised learning environment, the supervision signal can be a set of previously defined features that have to be detected to predict a class affiliation. It can also be the task of the algorithm to find features that represent the data. This type of learning is referred to as feature or representation learning. Representation learning is characterized by an algorithm that has to automatically learn feature representations to predict the labels of the dataset. This type of machine learning is more prominent in the unsupervised domain, as we often do not have labels to begin with.

An unsupervised learning algorithm only gets inputs X, intending to find patterns or other information within the data. One important example for the unsupervised learning of representations is clustering. Clustering algorithms have the goal of partitioning the dataset into groups with high intragroup similarity and low intergroup similarity [10] [18]. Clustering in general and a clustering algorithm, k-means, are further described in Section 2.7.

There are other subclasses like semi-supervised learning, where a learning algorithm can be partially supervised and partially unsupervised. Therefore a clear distinction between both of them is not always possible.

The term deep learning can be derived from the deep structure of the model. The model gathers knowledge from experience by learning complex concepts out of simpler ones, which are built on each other. Therefore many layers are used to represent these concepts, hence the name deep learning. [10]

After this short introduction to machine learning, we continue this Chapter by defining artificial neural networks. We then describe the training process as well as some structural additions to these types of models. The last Section of this Chapter covers traditional clustering approaches.

2.1 Artificial neural networks

Artificial neural networks, often just neural networks, are a composition of many units, called artificial neurons, which are inspired by biological neurons in the human brain. These units are typically functions, which map an input vector to a scalar. The input vector itself is a composition of function outputs of other units. A layer is a composition of many parallel acting units, which form a function, mapping a

vector to a vector. A neural network is therefore a framework, consisting of multiple chained functions. A basic type of neural network is the feedforward network. It takes an input x_n and learns to map it to a label y_n. The goal ist to approximate the optimal mapping $y_n = f^*(x_n)$ to a sufficient degree by learning the parameter θ of the function $y = f_\theta(x_n)$. The name arises from the nature of the model, where input data is feed through the network without information going back into the model. Therefore the function $f^{(i)}$ representing layer i can only get input values from previous functions $f^{(k)}, k < i$. Neural networks with information going back to the network are called recurrent neural networks. A network is called a deep neural network when it consists of multiple layers. The number of layers denotes the depth of the network. The last layer or output layer gives the approximated value of y_n. It is the only layer that is directly specified by the input x_n. The learning algorithm has the task to define the parameters of the other layers to best resemble the function f^*. These are called hidden layers. The distinction from what number of layers the neural network is called deep is not clearly defined [10].

2.1.1 Layer

Linear Layer

Linear layers are used as output units of the network. Let $h(x)$ be the output features of previous hidden layers. The linear layer performs an affine transformation to produce an output vector y that corresponds to the labels predicted by the network. It can be described by a matrix-vector multiplication of a weight matrix W and the addition of a bias vector b as the following equation:

$$z = W^\top h + b \tag{2.1}$$

Although this type of layer is usually used as an output layer, it can also be used as a hidden layer inside the network [10].

Activation

This layer can be described as an element-wise function $u(z)$, also called an activation function, applied to an affine transformation, which is the same as applying $u(z)$ to a linear layer.

$$u(z) = u(W^\top h + b) \tag{2.2}$$

Commonly used are rectified linear units (ReLu), which apply the function $u(z) = \max\{0, z\}$ as activation. In most cases, the activation function is chosen to be non-

linear, but linear functions can also be applied [10]. Another popular transformation is the sigmoid activation function:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

2.2 Loss Function

To optimize a learning function, we have the goal to minimize or maximize some error or loss measurement. These functions are called loss or cost functions and usually measure some distance between the approximated value and the real value. Some authors may use these terms separately, but we adopted the notation of Goodfellow et al. [10] to use both of them interchangeably. Finding the optimum of a loss function occurs through gradient-based optimization, which is further described in Section 2.4. First, we name some popular loss functions which find application in this and later Chapters.

2.2.1 Mean Squared Error

The mean squared error (MSE) is a simple and commonly used loss function. It uses the sum over the squared euclidean distance over every example of the training set divided by the number of training examples. First we define the euclidean distance or L^2 norm between two points $p = \{p_1, p_2, ..., p_M\}$ and $q = \{q_1, q_2, ..., q_M\}$ as:

$$d_{euc}(p, q) = \sqrt{\sum_{i=1}^{M}(p_i - q_i)^2} = \|p - q\|_2 \tag{2.4}$$

From there follows the squared euclidean distance by squaring each summand:

$$d_{sqe}(p, q) = \sum_{i=1}^{M}(p_i - q_i)^2 = \|p - q\|_2^2 \tag{2.5}$$

The MSE can be then written as the following equation, with N being the number of training examples, y_n the label of training example x_n, and $f_\theta(x_n)$ the estimated label for the training example:

$$MSE = \frac{1}{N}\sum_{n=1}^{N}\|y_n - f_\theta(x_n)\|_2^2 \tag{2.6}$$

2.2.2 Cross-Entropy

Let $p(y_n|x_n)$ be the probability of some function $f_\theta(x_n)$ predicting the corresponding label y_n. The cross-entropy loss is defined as:

$$CE = -\frac{1}{N}\sum_{n=1}^{N} \log p(y_n|x_n) \tag{2.7}$$

The probability distribution over a discrete space can be obtained by applying the softmax function to a class label prediction. The softmax function can be seen as a generalization of the sigmoid function σ [10]. It is defined as:

$$softmax(z_n) = \frac{e^{z_n}}{\sum_i e^{z_i}} \tag{2.8}$$

The cross-entropy is sometimes also referred to as the negative log-softmax function.

2.3 Backpropagation

During training, we are interested in the gradients of a loss function l with respect to all learnable parameters. The backpropagation algorithm [24] is an efficient way to let the information from the cost function go back through the network to calculate the gradients. It makes use of the chain rule of calculus to compute the gradient of a function from known gradients of other functions. The following illustration describes the chain rule applied to scalar functions. The rule is also applicable to the multidimensional case, where the input to a function is vector-valued and the derivates are calculated for every entry of the vector [10].

If we have a chained function $f(g(x))$, the derivative of f with respect to x can be written as equation 2.9.

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} \tag{2.9}$$

To make use of this property, the backpropagation algorithm stores the derivatives of f with respect to all other variables. Let $x = f(w)$, $y = f(x)$ and $z = f(y)$. We can calculate $\frac{\partial z}{\partial y}$ directly by taking the derivative of z with respect to y. The derivative with respect to x follows as:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} \tag{2.10}$$

If we want to calculate $\frac{\partial z}{\partial w}$, we can write this as equation 2.11 by also applying the chain rule.

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \tag{2.11}$$

We can now see that to arrive at equation 2.11, we only need to calculate the derivative of x with respect to w, as everything else has already been calculated. The same reasoning applies to 2.10 and $\frac{\partial z}{\partial y}$. The simple way of storing previously calculated derivatives makes backpropagation an efficient way to calculate the gradients needed to optimize the network.

The described algorithm only calculates the gradients. To optimize the network parameters, we make use of gradient-based optimization as described in the following Section.

2.4 Gradient-based Optimization

Recall from Section 2.2 that we want to find the minimal or maximal value of a loss function l. From now on, we only focus on the aspect of minimization because maximizing l can be rephrased as minimizing $-l$.

2.4.1 Gradient descent

Finding the minimum of a differentiable function analytically is straightforward in the sense that there are rules we can apply to find a local optimum. In machine learning, we often deal with functions that are not fully differentiable or otherwise complicated to evaluate. Therefore we have to use numerical computation to find the optimal values for the learnable parameters. The gradient descent algorithm uses step-wise optimization of the parameters θ of a function to derive at a local optimum or near a local optimum to minimize said function. This can be illustrated by using the definition of the difference quotient $\frac{f(x+\mu)-f(x)}{\mu}$. When μ approachs zero, the difference quotient becomes the derivative of $f(x)$ as descibed in equation 2.12.

$$f'(x) = \lim_{\mu \to 0} \frac{f(x+\mu) - f(x)}{\mu} \tag{2.12}$$

If μ is small enough we can approximate the change of $f(x)$ by solving equation 2.12 for $f(x+\mu)$:

$$f(x+\mu) \approx f(x) + \mu f'(x) \tag{2.13}$$

By replacing μ with $-\epsilon \operatorname{sign}(f'(x))$ we can clearly see that the new value of $f(x+\mu)$ is smaller then $f(x)$, thus moving x by a small amount into the direction of the negative derivative reduces the value of f. The same reasoning applies to the multidimensional case, where f depends on multiple variables and, instead of a

derivate, makes use of the gradient ∇f. Most gradient descent algorithms vary the variables of the function by the negative gradient multiplied with some small value ϵ. With $\nabla f(x)$ being the gradient of $f(x)$ with respect to the vector x, a gradient step can be described as:

$$x' = x - \epsilon \nabla f(x) \tag{2.14}$$

The small value ϵ is called the learning rate and determines the step size of a gradient step [10].

2.4.2 Stochastic Gradient descent

The training of neural networks often involves a lot of input data. Networks for image detection are often trained with millions of images. Calculating the gradient of the loss function for every input image would be computationally expensive. Thus other methods are deployed in practice, which only sample a subset of the data to compute the optimized weights of the network. Stochastic gradient descent (SGD), sometimes also referred to as mini-batch stochastic gradient descent, is the most prominent algorithm to achieve this goal. In every step of the algorithm, a minibatch is selected randomly from the training examples, which is then used to compute the gradients and perform the gradient step. In practical applications, the learning rate ϵ has to be decreased every other epoch to handle the noise introduced by sampling the minibatch. Even though SGD is much faster than using gradient descent on the whole dataset, it can still be a slow training process. To accelerate the training, some implementations make use of the concept of momentum. It is described by updating the parameters with a streaming average of previous gradients, which results in a bigger step size if successive gradients point in the same direction. Let $l_\theta(x_n, y_n)$ be a loss function depending on the parameters θ, inputs $x_n \in X$ and labels $y_n \in Y$, the gradient step with momentum can be described as equation 2.15. v corresponds to the streaming average, which can also be interpreted as the velocity at which the parameters change. Note that X and Y are not the whole dataset and corresponding labels, but only a minibatch of size b. $m \in [0, 1)$ is a momentum coefficient and b the mini-batch size [10].

$$v \leftarrow mv - \epsilon \nabla_\theta \frac{1}{b} \sum_{n=1}^{b} l_\theta(x_n, y_n) \tag{2.15a}$$

$$\theta \leftarrow \theta + v \tag{2.15b}$$

2.5 Batch Normalization

Optimization methods such as stochastic gradient descent accelerate the training process of a neural network, but as mentioned before, it can still be very slow. If a parameter changes during training, the distribution of the inputs to the next layer also changes. This leads to the need for lower learning rates and predefined parameters, resulting in a lengthy training process. Additionally, some activation functions tend to saturate during training, leading to a vanishing gradient and thus to problems in the optimization of those units. Let us consider the sigmoid activation function $\sigma(z)$. The derivative with respect to z becomes:

$$\sigma'(z) = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{e^{z}}{(1+e^{z})^2} \tag{2.16}$$

We can see clearly, that $\sigma'(z)$ approaches zero for high absolute values of z. Thus, training will proceed slowly, which is also amplified throughout the network, and the problem becomes worse with increasing depth. Batch normalization [15] (BN) reduces these issues by normalizing the input to a layer. Each feature gets normalized independently. Let x_n^k be the k-th feature of the input example x_n. Normalization occurs through the following equation, with $E[x^k]$ and $Var[x^k]$ being the mean and variance of the k-th feature of the whole training set:

$$\hat{x}_n^k = \frac{x_n^k - E[x^k]}{\sqrt{Var[x^k]}} \tag{2.17}$$

This could reduce the expressive power of the network, which is addressed through the use of a shifting parameter γ^k and a scaling parameter β^k, resulting in the following equation:

$$y_n^k = \gamma^k \hat{x}_n^k + \beta^k \tag{2.18}$$

Though most networks use a mini-batch-based optimization strategy, batch normalization can also be applied to this setting. Algorithm 1 shows the process of calculating the normalized output. K is the number of feature dimensions or the number of units in the layer and ϵ is a small constant that is added for numerical stability [15].

Algorithm 1 Batch Normalization applied to a mini-batch

Input: mini-batch dataset $X = \{x_1, .., x_b\}$
Output: normalized output features $Y = \{y_1, .., y_b\}$
for $k \in \{1, ..., K\}$ **do**
    $\mu \leftarrow \frac{1}{b}\sum_{n=1}^{b} x_n$ ▷ mini-batch mean
    $\sigma \leftarrow \frac{1}{b}\sum_{n=1}^{b}(x_n - \mu)^2$ ▷ mini-batch variance
    for $x_n \in X$ **do**
        $\hat{x}_n^k \leftarrow \frac{x_n^k - \mu}{\sqrt{\sigma - \epsilon}}$ ▷ normalize
        $y_n^k \leftarrow \gamma^k \hat{x}_n^k + \beta^k$ ▷ scale and shift
    end for
end for

2.6 Convolutional Neural Network

Another type of neural network is the convolutional neural network (CNN). A CNN is a neural network that utilizes a mathematical operation named convolution. It is used in applications where the input data has a kind of grid structure like the processing of images or time series. The mathematical definition of the convolution operation of two absolutely integrable functions $x(t)$ and $w(t)$ is:

$$s(t) = (x * w)(t) = \int_{-\infty}^{\infty} x(u)w(t-u)\mathrm{d}u \tag{2.19}$$

From there, the definition of the discrete convolution follows as:

$$s(t) = (x * w)(t) = \sum_{u=-\infty}^{\infty} x(u)w(t-u)\mathrm{d}u \tag{2.20}$$

In the context of machine learning $x(t)$ can be seen as the input and $w(t)$ as a weighting function, also called the kernel. In machine learning, both the input and the kernel are often described by multidimensional matrices of data entries and learnable parameters, respectively. Equation 2.20 can therefore be written as equation 2.21, where $S(i,j)$ denotes the output matrix at position (i,j), I the input matrix and K the kernel.

$$S(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.21}$$

$$= \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.22}$$

The convolution operation is commutative, which leads to equation 2.22. It is a little more simple to implement equation 2.22 due to less variation in the range of m

and n. The commutative characteristic can be seen as flipping the kernel relative to the input so that the decreasing and increasing indices are exchanged. Not flipping the kernel yields the following equation, called the cross-correlation:

$$S(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.23}$$

It is also called convolution without flipping the kernel in the context of neural networks. Which operation to use in the concrete application is not important, as the parameters of the kernel have to be learned in both cases. A convolutional layer in a neural network typically consists of more than one kernel to extract multiple different features [10].

There are some benefits in using a CNN over a standard neural network. A CNN uses sparse interactions between units. While the first layers of the network are connected to only a couple of input units, the deeper layers could be linked to more of them by indirect connectivity from higher layers. This is achieved by deploying a kernel that is smaller than the input matrix. It leads to an improvement in memory usage and better statistical efficiency. The sharing of parameters further improves the named benefits. Compared to a normal layer which uses matrix multiplication, where a weight matrix with $N \times M$ parameters has to be learned, the kernel only has a fixed number of K parameters, with N being the number of inputs, M the number of outputs, and $K \leq N$ the number of kernel entries. Another benefit is the equivariance to translation. This means that for translative transformations like shifting every input pixel one unit to the right, the convolution can be applied before or after the transformation, resulting in the same output. Convolution reduces the width of the input data by $K+1$, which leads to a smaller representation of the data at each layer. It can be circumvented by reducing the kernel size, also leading to a less informative representation. To control both the number of layers and the size of the kernel independently, zero padding is used. By adding zeros to the borders of the input, the size of the output can be the same as the input or larger [10].

2.6.1 Pooling Layer

Another type of layer, often used in combination with convolutional layers, is the pooling layer. Pooling replaces the output unit with a summary of multiple other units within some defined neighborhood, where the dimension remains the same. A popular pooling function is max pooling, where the function outputs the maximum inside a rectangular region around the current unit. Aside from the maximum,

other functions are used like the average value or the L^2 norm, shown in equation 2.4. The pooling layer leads to an output that is invariant to small input changes. It is a useful property when we are interested in the presence of a feature but not its exact location. Combining the pooling layer with the convolutional layer leads to further improvements in terms of transformation invariance. Often, the convolution is carried out with multiple kernel matrices to get different representations of the input data. In combination with a pooling layer, a neural network can learn to be invariant to larger transformations of the input. Applied to image data, this could mean an invariance to rotations of an image. It is also possible to output fewer units than inputs given to the pooling layer by only mapping the input onto every k-th unit. k is called the stride between two output units, with $k = 1$ leading to the same input and output size. Pooling layers are also beneficial when the input is not of a fixed size. The input can therefore be mapped to a fixed smaller dimension, by a variation in the pooling stride and width, with the width being the number of inputs influencing one output unit [10].

2.7 Traditional Clustering Approaches

As mentioned at the beginning of this Chapter, the objective of a clustering algorithm is to partition a dataset into subsets or clusters, where similar data points are in the same cluster. This type of clustering is known as centroid-based clustering. In centroid-based clustering methods, each cluster is represented by a centroid or cluster center μ, typically the mean of all assigned data points. It can be described as the following optimization problem, where x_n denotes a datapoint, $d(p, q)$ a distance measure, and μ_n the nearest centroid to the datapoint x_n.

$$\forall x_n \in X, \text{ minimize } d(\mu_n, x_n) \tag{2.24}$$

In the following two paragraphs, we will describe the most widely used centroid-based clustering method k-means and a variation of the k-means algorithm mini-batch k-means.

2.7.1 K-means

The k-means algorithm initializes the centroids of k clusters with random values. It then assigns every training example x_n to its nearest centroid μ, which is determined by a distance measure d_{sqe}, the squared euclidean distance.
After all x_n are assigned to a cluster, the centroids are calculated as the mean of all assigned examples. This process is continued until convergence is reached, which means the cluster assignments do not change anymore [10]. The pseudo-code for k-means can be seen in Algorithm 2.

Algorithm 2 k-Means

Input: dataset X, number of clusters k
Output: k clusters
for $\mu \in M$ **do** ▷ Initialize all centroids μ
    $\mu \leftarrow$ random value in range of training examples
end for
repeat
    for $x_n \in X$ **do**
        assign each x_n to nearest cluster μ
    end for
    for $\mu \in M$ **do**
        $\mu \leftarrow$ mean of all assigned training examples
    end for
until convergence is reached

2.7.2 Mini-batch k-means

There are stochastic gradient descent based algorithms, which reduce computation time but also produce lower quality clusters. An algorithm that reduces computation time by several orders of magnitude and also preserves cluster centers is mini-batch k-means [25]. Mini-batch k-means clusters the given inputs by forming mini-batches, randomly drawn from the dataset in every iteration. The next step is analog to k-means, where each sample is assigned to its nearest centroid. The centroids are then updated on a per-sample base, which further reduces convergences time [25]. Algorithm 3 gives a more detailed description of mini-batch k-means.

Algorithm 3 Mini-batch k-Means

Input: number of clusters k, mini-batch size b, iterations t, data set X
Output: k clusters
for $\mu \in M$ **do** ▷ Initialize all centroids μ
    $\mu \leftarrow x_n$ randomly picked from X
end for
$v \leftarrow 0$
for $i = 1$ to t **do**
    $S \leftarrow b$ examples randomly picked from X
    for $x_n \in S$ **do**
        $d[x_n] \leftarrow \underset{\mu \in M}{arg\,min}\ d_{euc}(\mu, x_n)$ ▷ Cache the center nearest to x_n
    end for
    for $x \in S$ **do**
        $c \leftarrow d[x_n]$ ▷ Get cached center for this x
        $v[c] \leftarrow v[c] + 1$ ▷ Update per-center counts
        $\eta \leftarrow \frac{1}{v[c]}$ ▷ Get per-center learning rate
        $c \leftarrow (1 - \eta)c + \eta x_n$ ▷ Take gradient step
    end for
end for

Chapter 3

Methods

3.1 Deep Clustering

In [4], Caron et al. propose Deep Cluster (DC), a method to learn a convolutional neural network with labels found by a standard clustering algorithm. To achieve this, the algorithm iteratively finds clusters of features and uses these to update the weights of the neural network.

The network is learned similarly to supervised training, where the goal is to find good general-purpose features. In supervised training, the features are learned through a mapping function $f_\theta(x_n)$. To predict the correct labels from f_θ, a classifier g_W is used on top of the extracted features. To learn the parameters W and θ together, the cost function in equation 3.1 has to be minimized.
To initialize the network, the parameters θ are sampled from a Gaussian distribution. Although the produced features are not good, the accuracy of the network with random weights is above the chance level. The features are then clustered by k-means to derive pseudo-labels y_n for the next step. These labels are used to minimize the cost function, which uses the negative log-softmax function l. Minimization occurs through the use of mini-batch stochastic gradient descent and backpropagation.

$$\min_{\theta,W} \frac{1}{N} \sum_{n=1}^{N} l(g_W(f_\theta(x_n)), y_n) \tag{3.1}$$

The training consists of alternating between the clustering and minimization of the cost function until the clusters stabilize. A fraction of images are still constantly reassigned to other clusters, but this has no impact on the training.
It exists several trivial solutions, which are caused mainly by empty clusters and trivial parametrization. To prevent empty clusters, a non-empty cluster is selected

randomly whenever an empty cluster occurs. The previous empty cluster gets a new centroid based on the other cluster's centroid with a random small change. Then all points of the non-empty cluster are assigned between both clusters based on their nearest centroid. The other problem of trivial parameterization occurs whenever the majority of images are assigned to very few clusters. The worst-case would be every input getting the same predicted output. It is avoided by sampling images based on a uniform distribution between pseudo labels. The same effect would be achieved by adapting the loss function in equation 3.1 so that each contribution of an input is re-weighted proportional to $\frac{1}{N_c}$. N_c denotes the number of assigned samples to cluster c [4].

3.2 Online Deep Clustering

Online Deep Clustering (ODC) [27] is very similar to Deep Clustering, by using the same approach of representation learning and clustering, but instead of alternating between these two tasks, Online Deep Learning updates the labels and network parameters rather continuously. ODC introduces two dynamic memories. One is for centroids, which are the mean of all assigned samples, similar to a standard clustering algorithm. The other one is a memory for samples, storing extracted features and pseudo-labels for the whole dataset. The memories are initialized by a global clustering method, k-means. The function f_θ, which is used for feature extraction, and the linear classifier g_W are initialized randomly. The first step is to extract the features from the input samples. This works in the same way as in Deep Clustering. Then the network parameters are updated through mini-batch stochastic gradient descent with the pseudo-labels stored in the sample memory. This is done through minimization of the same cost function Deep Clustering uses. The optimization problem follows therefore equation 3.1. Next, a gradient step is performed for every sample to update the features in the sample memory. It is described by the following equation, where $F(x_n)$ denotes the feature stored in the memory corresponding to sample x_n, $m \in (0, 1]$ is a momentum coefficient.

$$F(x_n) \leftarrow m\frac{f_\theta(x_n)}{\|f_\theta(x_n)\|_2} + (1-m)F(x_n) \tag{3.2}$$

At the same time, each sample gets a new label based on its nearest cluster centroid using the squared euclidean distance d_{sqe} following equation 2.5. Lastly, the centroids are updated through clustering and stored in the centroid memory, this occurs only every k-th iteration, with k being predefined by the user.

ODC also needs to address trivial solutions like all clusters collapsing into some huge clusters or trivial parametrization. Uniform sampling between pseudo-labels would not be suitable for the method because it would require to re-sample the entire dataset in each iteration, which is deemed costly and redundant by the author. Another approach is therefore used. As already stated in [4], the loss function in equation 3.1 can be altered, by re-weighting the loss. The authors tested a re-weighting proportional to $\frac{1}{\sqrt{N_c}}$ on Deep Clustering, which held the same results and was therefore adopted. To prevent empty clusters, a small cluster $c_s \in C_s$, which is determined by a given threshold, is cleared by assigning all samples to its nearest centroid of the remaining clusters $C \setminus C_s$. Further, the largest cluster, $c_m = \max\{C \setminus C_s\}$ is split into two clusters by applying k-means. One of the clusters is then randomly assigned as the new cluster c_s. This is repeated until all cluster sizes are greater than the given threshold [27].

3.3 Self-Labeling

The third algorithm is called Self-Labeling via simultaneous clustering and representation learning (SeLa) [2]. SeLa also alternates between learning a neural network with pseudo labels and creating these pseudo labels, but adopts a slightly different strategy from the first two algorithms. Both previously described approaches minimize one function for the neural network and one for the clustering algorithm. The goal of SeLa is to minimize only one cost function instead of two. This could lead to a solution where all training examples are assigned to only one cluster. To solve this issue, Asano et al. added a restriction that causes the labels to be equally partitioned between all classes. To get a more clear formulation of the problem, equation 3.1 can be rewritten as equation 3.3 to include a concrete loss function, the cross-entropy loss from equation 2.7.
$p(y_n|x_n)$ is the class probability of $g_W(f_\theta(x_n))$ predicting y_n, which is obtain through application of the softmax function (equation 2.8) to g_W [2].

$$E(p, y_1, ..., y_N) = -\frac{1}{N}\sum_{n=1}^{N} \log p(y_n|x_n) \tag{3.3}$$

The next step is to encode the labels as posterior distribution $q(y|x_n)$ which results in equation 3.4, with K being the number of distinct labels.

$$E(q,p) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{y=1}^{K} q(y|x_i)\log p(y|x_i) \tag{3.4}$$

With the restriction of equally divided classes, the overall optimization problem can be written as equation 3.5.

$$\min_{p,q} E(p,q) \quad \text{subject to} \quad \forall y : q(y|x_n) \in \{0,1\} \text{ and } \sum_{n=1}^{N} q(y|x_n) = \frac{N}{K} \tag{3.5}$$

It can be seen as an optimal transport problem, which is solved by a fast version of the Sinkhorn-Knopp [7] algorithm.

3.4 Other Related Work

The work of Buschjäger et al. [3] is closely related to the objective of this thesis, as they use the same dataset to conduct their experiments and aim at finding a good performing model for gamma hadron separation. The main focus is training a deep neural network and a binary neural network to compare these to the currently deployed machine learning pipeline. Their findings show that deep models trained on the raw photon counts perform significantly better, in terms of detection accuracy on labeled data, than the state-of-the-art random forest model, which uses hand-crafted features for detection. They also conduct the $S_{Li\&Ma}$ significance test on the real data taken by the FACT telescope, which is further explained at the end of Chapter 6.

Chapter 4

Dataset

Detecting gamma rays from space is an important research area inside the field of astrophysics. Celestial objects can be recognized by observing their emitted energy. Cosmic gamma rays do not travel up to the surface of the earth. They scatter into subatomic particles, which can again react with other particles to form a cascade-like structure. Particles inside these structures travel at high velocities and emit visible light, known as Cherenkov light. These particles, however, can be detected by Cherenkov telescopes through large reflectors and high-speed cameras. [6]

The images we want to investigate were taken with the FACT telescope [1], which consists of 1440 hexagonal detector plates. If a detector plate detects a specific amount of energy, the camera of the telescope starts to record the light pulses. Each image amounts to a time sequence of 150 nanoseconds and allows for a reconstruction of the energy and direction of the particles creating the Cherenkov light.

The telescope can operate in an additional mode, called the 'Wobble' mode, which uses the false source tracking method [9]. The camera gets oriented so that the assumed gamma source is placed on a circle around the camera center. The surrounding region is called the *On* region and aims at measuring the energy of the gamma rays emitted by the source. All other regions on the circle are called *Off* regions and can measure the background radiation, as long as these regions are not directed at a gamma source. The 'Wobble' mode gets his name from the periodical changes of the *On* region [21]. As the earth rotates along its axis and moves through space, we perceive it as the gamma source moving on a fixed trajectory, making these periodical changes of *On* and *Off* regions necessary.

We make use of the publicly available real-world observations taken from the crab nebula, a known gamma source [1]. We use the same dataset as Buschjäger et al. [3] used in their work, which had already been pre-processed for the use of machine learning. The sensors had to be calibrated to correct for environmental interferences. The images were reduced to a time series of 50 nanoseconds, focusing on the most relevant part of the time series. They derived a baseline measurement consisting of the energy of one photon hitting the sensors and subtracted this baseline as often as possible from the actual image to obtain a photon count for each pixel. The last preprocessing step was transforming the hexagonal grid into a rectangular grid to use common neural network architectures. The resulting dataset consists of 3.972.043 images with a shape of 45 by 45. 757.993 of these images were taken in the 'Wobble' mode. An image of a detected particle in the original hexagonal shape can be seen in Figure 4.1.

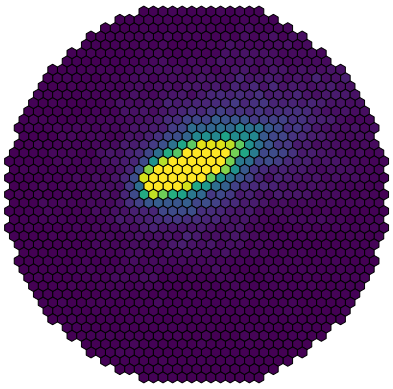
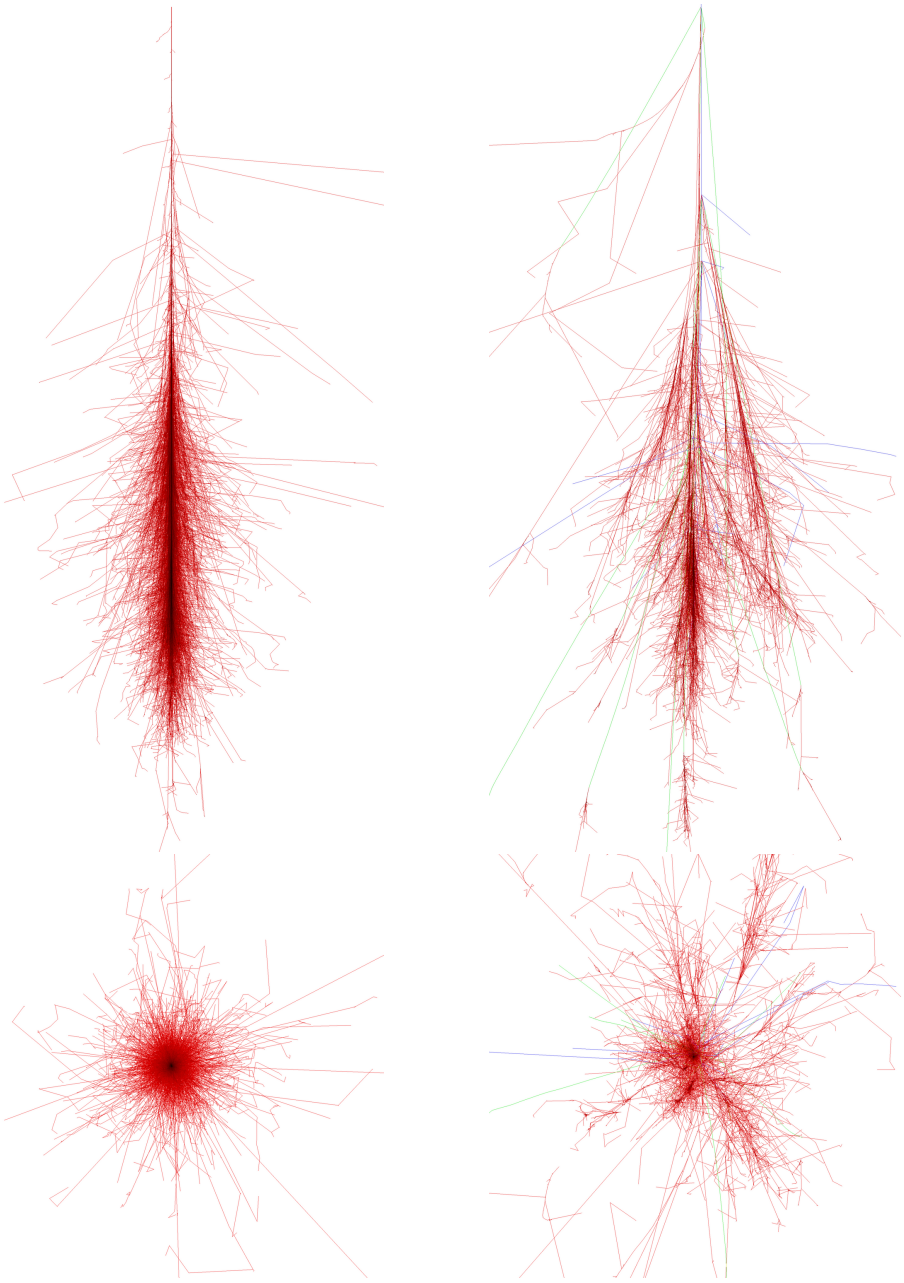


Figure 4.1: Image of a detected particle

Often we prefer labeled data, as supervised machine learning algorithms rely on some kind of ground truth to make predictions or learn a feature representation. As particles detected by a telescope come without labels, we desire a different kind of labeled dataset. We make use of a simulated dataset, which is created by the particle simulation software CORSIKA (Cosmic Ray Simulations for Kascade) [12]. CORSIKA uses a Monte Carlo simulation in combination with profound knowledge of high energy and electromagnetic interactions. To create realistic data, the simulation software creates particle showers, which are then run through a simulation of the telescope. A visualization of the simulated particle showers can be seen in Figure 4.2. The resulting images are of the same shape as the real images, with an equal distribution between gamma rays and background radiation. The data comes

in two variations, "Diffuse" and "Wobble". In Chapter 6, where we conduct our experiments, we mainly focus on the Diffuse dataset. We refer to it simply as the simulation dataset and explicitly mention the use of the Wobble data. The simulated data also comes in a version with so-called quality cuts. This means the developer of the software filtered the dataset for images that do not resemble real live events. We make no use of these quality cuts and train our models with the uncut dataset. This leaves us with a total image count of 300.000, split into a training set with 200.000 and a test set with 100.000 images. The Dataset is provided to us in the NumPy [11] format, which leaves us with a threedimensional array, with an entry for every image.



(a) Gamma Shower (b) Proton Shower

Figure 4.2: Images of two simulated extensive air showers, taken from [8]. They are both induced by a primary particle with an energy of 100 GeV. In the left image the primary particle is a gamma particle, in the right image a proton. The colored tracks correspond to the particles: Red: electrons, positrons or gamma, Green: muons, Blue: hadrons

Chapter 5

Implementation

In this Chapter, we will briefly discuss the implementation process. The code for the experiments was written in the programming language Python. The main additional libraries used in the implementation process are the open-source deep learning framework PyTorch [19], NumPy [11] and the machine learning library scikit-learn [20]. For the neural network that had to be trained for the clustering methods, we used the implementation of a small VGG-type model by Buschjäger et al. [3], which was slightly modified to suit the task at hand better. The concrete architecture is described in the next Chapter. The algorithms described in Chapter 3 had already been implemented by their respective authors. This thesis uses their published implementations.
In the following paragraphs, we summarize the changes that had to be made to the implementations in order to successfully apply them to our data.

Deep Cluster The Deep Cluster implementation[1] had to be updated because it used an older version of Python and Pytorch. The framework of the neural network had to be adapted only structurally. The main difficulty was the dataset. The first approach was to load the NumPy array and convert it into a PyTorch Tensor. This worked fine for a small enough dataset. While trying to train the whole dataset with nearly 4 million images, memory issues occurred. The solution to the problem was only to load a mapping of the dataset. Thankfully NumPy arrays can be directly loaded as a memory map. This method to load the dataset led to another problem, the freezing of the training process after a couple of epochs, which could be circumvented by setting the number of workers to load the data to 0.

[1] https://github.com/facebookresearch/deepcluster

SeLa The SeLa implementation[2] was structured very similar to Deep Cluster so that most of the changes like the dataset and neural network could be adopted. One problem in the implementation led to a very slow running time when only one graphic processing unit (GPU) was used. Less than two GPUs always resulted in the operations being carried out on the central processing unit (CPU). The program was not written for only one GPU and had to be manually adapted.

Online Deep Cluster For Online Deep Cluster, the authors provided a library containing multiple unsupervised representation learning algorithms[3]. This led to some issues with the adaptability of the algorithm, mainly because of readability issues, with methods being separated into a lot of subclasses. Again the dataset and the neural network could be used with some changes. The neural network had to be separated into smaller parts. For example, the feature extraction pipeline was separated from the classification head and had to be initialized separately. The core features of the dataset could be adopted, but multiple other files had to be adapted, which had to do with the way we load the data and the structure of the library. Similar to Deep Cluster, the ODC implementation had some conflicts with the dataset, which was also solved by setting the data loader workers to zero.

Overall the implementations of Deep Cluster and SeLa were written clearly, while the Online Deep Cluster implementation was more difficult to comprehend, mainly because of the additionally used library, which handled a lot of the training process. For k-means and mini-batch k-means we used the implementation from scikit-learn [20].

[2] https://github.com/yukimasano/self-label
[3] https://github.com/open-mmlab/OpenSelfSup

Chapter 6

Experiments

The main focus of this thesis is the examination of training methods involving unsupervised representation learning. The in Chapter 3 proposed algorithms use feature clustering to derive pseudo-labels for the training process of a deep neural network. To generally examine these approaches and to compare the different algorithms, we first need to define an assessment strategy. In general, evaluating unsupervised machine learning methods is difficult because we do not have labels describing the data or some other ground truth. We make use of the trained networks through supervised re-learning of the classifier with a subset of the labeled simulation data. Therefore the parameters of the layers used for feature extraction stay fixed. Another way to analyze the methods is by the evaluation of the resulting cluster assignments.
Before we further discuss methods of comparison, we display the architecture of the neural network and talk again briefly about the dataset.

Architecture The architecture of our deep neural network is a small VGG type model. It was chosen for two main reasons. One being the improved results reported by Caron et al. [4] over other popular types of architectures. The other main reason was the previous work from Buschjaeger et al. [3], which also used this type of model and was provided for our experiments. As already stated, we modified the provided implementation of the neural network slightly to better adapt to the given models.

The first three layers consist of one convolutional layer with a kernel size of 3 and a stride of one, one batch normalization layer, and one rectified linear unit as activation layer. We repeat these three layers followed by one maximum pooling layer with a kernel size of 2 and a stride of 2. This structure is what we will call blocks for the rest of this chapter. The network consists of two of these blocks for feature

extraction. The classifier consists of two linear, a batch normalization, and two activation layers. The final layer is a linear layer, which is adapted to the number of clusters we wish to train for. The architecture for a block size of two can be seen in Table 6.1, which is what we used for most experiments after we found the best hyperparameters.

Layer	Output Size	Kernel Size	Stride
Convolutional	$16 \times 45 \times 45$	3×3	1×1
Batch Normalization	$16 \times 45 \times 45$	–	–
Activation	$16 \times 45 \times 45$	–	–
Convolutional	$16 \times 45 \times 45$	3×3	1×1
Batch Normalization	$16 \times 45 \times 45$	–	–
Activation	$16 \times 45 \times 45$	–	–
Max Pooling	$16 \times 22 \times 22$	2×2	2×2
Convolutional	$32 \times 22 \times 22$	3×3	1×1
Batch Normalization	$32 \times 22 \times 22$	–	–
Activation	$32 \times 22 \times 22$	–	–
Convolutional	$32 \times 22 \times 22$	3×3	1×1
Batch Normalization	$32 \times 22 \times 22$	–	–
Activation	$32 \times 22 \times 22$	–	–
Max Pooling	$32 \times 11 \times 11$	–	–
Linear	512	–	–
Batch Normalization	512	–	–
Activation	512	–	–
Linear	512	–	–
Activation	512	–	–
Linear	number of clusters	–	–

Figure 6.1: Architecture of the network

Dataset As we already described the dataset in detail in Chapter 4, we will briefly name how we split our dataset for the following experiments. We further divide the simulated training set into a smaller set consisting of 100.000 images, which is mainly used for training during the hyperparameter search. The other half of the images

are used for the re-learning process of the classification head. We use 80.000 images for training the classifier and 20.000 for validation. It is further denoted as the label-mapping set.

The rest of this Chapter is organized in the following way: First, we derive the hyperparameters for the training process. Then we train our models with the entire FACT dataset and compare their accuracy using the simulation data. In the third experiment, we compare the resulting clusterings of the models as well as k-means and mini-batch k-means. This is followed by an experiment, where we train a supervised neural network to compare it directly to our unsupervised algorithms. In the last experiment, we evaluate our models on the FACT dataset, using the Li & Ma significance test [16]. Additionally, we provide a visualization of some gamma and hadron images, as predicted by our methods.

6.1 Hyperparameter

Hyperparameters are parameters of the model, which can not be found during training. The search for those parameters can be a tedious process because there exist infinitely many options to choose from. The approach taken in this thesis is a combination of an extensive search and manual search. First, we have to focus on which hyperparameters we want to optimize. The structure of the neural network alone is one large parameter that has to be found. Not all neural network architectures perform equally well on different problems. To focus on the more important aspects of this thesis, we use a predefined neural network as described at the beginning of this Chapter and vary only the number of blocks. As mentioned in [4], some amount of over-segmentation is beneficial for the learning process, so we adopt the same approach in testing multiple numbers of clusters. The last parameter to find is the learning rate, which probably has the most significant influence on the training results as it controls the influence of the gradient steps taken in every iteration. We decided to take a smaller subset consisting of 100.000 images randomly sampled from the whole simulation training set to test for hyperparameters. One reason not to train the network on real images from the crab nebula is that we do not know the dataset's distribution of gamma and non-gamma events. This could lead to a bias in the training process and worse detection rates of gamma or hadron particles. Another reason is to use the so trained models again for the supervised experiment in Section 6.4, where training with real data is not possible because of missing labels.

The first training run for each algorithm consisted of varying the block size and the learning rate. We test 2,3,4, and 5 blocks and learning rates ranging from 0.1 to 0.00001. The number of clusters was held fixed at 32 clusters, which initially gave a good accuracy. We evaluate the models by re-learning the classifier as mentioned at the beginning of this Chapter and test the accuracy with 100.000 images of the test dataset. Table 6.1 shows the results of the first search.

Learning Rate	Blocks	DC	ODC	SeLa
0.1	2	64.983	69.082	68.199
0.1	3	**67.556**	67.94	67.487
0.1	4	64.393	64.651	64.964
0.1	5	59.63	58.834	58.229
0.01	2	64.436	**69.672**	66.34
0.01	3	65.827	67.887	67.06
0.01	4	63.628	65.517	65.738
0.01	5	62.707	61.775	60.096
0.001	2	65.86	68.839	**70.013**
0.001	3	63.751	66.913	67.842
0.001	4	60.562	66.094	66.018
0.001	5	62.692	59.843	61.204
0.0001	2	65.669	67.684	69.284
0.0001	3	67.408	66.788	69.138
0.0001	4	63.822	65.33	66.898
0.0001	5	59.33	60.599	60.399
0.00001	2	64.835	65.529	67.649
0.00001	3	65.765	66.044	65.22
0.00001	4	60.921	60.9	61.952
0.00001	5	57.406	57.898	57.29

Table 6.1: The first hyperparameter search results

As can be seen in Table 6.1, the best results for all algorithms were obtained with a lower number of blocks. In the next search, we included a block size of two and three. Even though most of the results favored a size of two, three blocks performed comparably well. We varied the number of clusters in a range of 2 to 256 and tested

the same learning rates as before. The values obtained from the final hyperparam-

Learning Rate	Clusters	DC	ODC	SeLa
0.001	16	66.397	**69.893**	68.319
0.001	32	67.744	69.127	**69.554**
0.001	256	**70.167**	68.114	68.62

Table 6.2: Selected accurcacys for 2 blocks

eter run can be seen in Table 6.2. The best values are marked in their respective column. The full table for 2 blocks and 3 blocks can be found in Table A.1 and Table A.2 in the Appendix A.

In [2], the authors proposed that using multiple heads for classification could lead to an improvement in test accuracy. We tested the use of multiple heads varying from one to ten. The results suggest no further improvement. Therefore the strategy was not adopted in the other two models. Further tested was the influence of training more epochs than 50, which also resulted in slightly worse accuracy, probably due to overfitting.

6.2 Training on FACT Data

The next experiment evaluates the performance of the models on real training data recorded with the FACT telescope. The original papers trained their networks for more than 400 epochs on datasets like ImageNet. Their training set consists of more diverse images of higher resolution. As our dataset contains nearly 4 million images, we choose to train for fewer epochs. We trained for 50 and five epochs, with the best parameters obtained by the hyperparameter search. For the label mapping, we used the same dataset as before. The test results can be seen in Table 6.3 and were also evaluated on the test dataset consisting of 100.000 images.

Epochs	DC	ODC	SeLa
50	64.51	67.319	68.974
5	67.811	**69.308**	68.985

Table 6.3: Accurcacy for training on the full FACT dataset

We obtained the best accuracy on Online Deep Cluster trained for only five epochs. The results depict a better accuracy for all models when trained only for a few epochs. As we reported worse accuracy for models trained for more than 50 epochs in the hyperparameter training with only 100.000 images, it is not surprising that the accuracy of the models is slightly worse when we train with much more images as the models can overfit to the training data. This can most evidently be observed in Deep Cluster and Online Deep Cluster, while Sela remains relatively stable.

6.3 Cluster Comparison

As mentioned in Chapter 1, one important experiment is the comparison of the proposed methods to a traditional clustering approach. As traditional methods such as k-means make no use of an underlying neural network, we will examine the resulting clusters of each method. We start by presenting some criteria for cluster evaluation.

6.3.1 Metrics

Rand Index The Rand Index (RI) [22] is named after William M. Rand. He extends the idea of a simple count of misclassified points against a correct classification to how pairs of points are clustered. Let a and b be two different data points of the dataset. Two clusterings C and D can be seen as similar either if a and b are clustered together in both C and D or if they are in different clusters in both C and D. Dissimilarity in this sense means that a and b are in the same cluster in one clustering and in different clusters in the other clustering. The Rand Index is given by the sum over all similar pairs, divided by the number of possible pairs, with γ_{ij} being equal to 1 for every similar pair and 0 for every dissimilar pair:

$$RI = \frac{\sum_{i<j}^{N} \gamma_{ij}}{N\binom{N}{2}} \tag{6.1}$$

A RI close to zero means that clusterings are very dissimilar and a RI of one means they are the same.

One disadvantage of the Rand Index is that it is not near zero for a random clustering. This issue can be addressed by also using the adjusted Rand Index [13], which is adjusted for chance. It can be derived by a general form of index correction

following equation 6.2.

$$AdjustedIndex = \frac{Index - ExpectedIndex}{MaxIndex - ExpectedIndex} \tag{6.2}$$

Mutual Information The uncertainty of a discrete random variable X can be measured by the entropy $H(X)$, which is defined as:

$$H(X) = -\sum_{n=1}^{N} p(x_n) \log(p(x_n)) \tag{6.3}$$

$p(x_n)$ is the probability of $X = x_n$. The mutual information of two random variables X, Y with a joint probability distribution $p(x_n, y_m)$ is defined by equation 6.4 [5].

$$MI(X,Y) = -\sum_{n=1}^{N}\sum_{m=1}^{M} p(x_n, y_m) \log\left(\frac{p(x_n, y_m)}{p(x_n)p(y_m)}\right) \tag{6.4}$$

We can write the entropy of a Clustering C as:

$$H(C) = -\sum_{i=1}^{S} P(i) \log(P(i)) \tag{6.5}$$

The probability that a sample from the whole dataset with size N falls into cluster C is given by: $P(i) = \frac{|C_i|}{N}$, where $|C_i|$ is the number of samples in the cluster. The Mutual Information of two clusterings C and D is therefore given by:

$$MI(C,D) = -\sum_{i=1}^{S}\sum_{j=1}^{R} P(i,j) \log\left(\frac{P(i,j)}{Pi)P(j)}\right) \tag{6.6}$$

$P(i,j) = \frac{|C_i \cap D_j|}{N}$ is the probability of one sample belonging to cluster C_i in C and cluster D_j in D [26].

The Mutual Information is as well as the Rand Index not adjusted to chance and has the additional disadvantage that comparing two equal clusterings does not result in a score of one. We use the adjusted mutual information metric [26], which uses the same index correction as the adjusted Rand Index.

V-Measure The V-measure [23], where V stands for validity, measures the harmonic mean between a homogeneity score and a completeness score. For both homogeneity and completeness, we need the conditional entropy $H(Y|X)$:

$$\begin{aligned} H(Y|X) &= -\sum_{n=1}^{N}\sum_{m=1}^{M} p(x_n, y_m) \log(p(y_m|x_n)) \\ &= -\sum_{n=1}^{N}\sum_{m=1}^{M} p(x_n, y_m) \log\left(\frac{p(x_n, y_m)}{p(x)}\right) \end{aligned} \tag{6.7}$$

This can be written in terms of our clustering problem, with $P(i,j)$ and $P(i)$ being defined as in equation 6.6:

$$H(Y|X) = -\sum_{i=1}^{S}\sum_{j=1}^{R} P(i,j)\log\left(\frac{P(i,j)}{P(i)}\right) \tag{6.8}$$

After Rosenberg and Hirschberg [23], a clustering is homogeneous if only those data points are assigned to a cluster, whose underlying classes are the same. A clustering is complete if all data points of the same underlying class are assigned to the same cluster. With the conditional entropy, a clustering C a predefined class distribution K, homogeneity and completeness are defined as

$$hom(C,K) = 1 - \frac{H(K|C)}{H(K)} = \frac{H(K) + H(C) - H(C,K)}{H(K)} \tag{6.9}$$

and

$$com(C,K) = 1 - \frac{H(C|K)}{H(C)} = \frac{H(C) + H(K) - H(C,K)}{H(C)} \tag{6.10}$$

With β being a weighting factor, the V-measure can thus be defined as:

$$V = \frac{(1+\beta)hom \times com}{\beta \times hom + com} \tag{6.11}$$

We leave β at a value of one, giving both scores the same weight.

6.3.2 Comparison

As previously mentioned, the real images obtained by the FACT telescope come without labels. For this reason, we compare the methods on the simulation data. We train each model as well as k-means and mini-batch k-means with the full simulation dataset and varying numbers of clusters. We test two, 16, 32, and 256 clusters for each method. As the (adjusted) Rand Index, the (adjusted) Mutual Information score, and the V-measure, are all symmetrical, we can also use these metrics to compare two obtained clusterings from different methods against each other. Note that both the Rand Index and the V-measure range from 0 to 1, while the adjusted RI and the adjusted MI score have both a range of -1 to 1. We include all these metrics in our evaluation because they all value different aspects to judge a good clustering. A good clustering should therefore not only give a good score on one metric. For the following tests, we use the implementations of the metrics from scikit-learn [20].

First, we compare the resulting clusters to the labels of the simulation. As the scores for 2, 16 and 32 clusters gave similar results, we only display the obtain scores for 256 clusters. They can be seen in Table 6.4.

256 Clusters	Rand Index	adjusted RI	adjusted MI	V-measure
Deep Cluster	0.50022	0.00044	0.00421	0.00442
Online DC	0.50013	0.00026	0.00382	0.00392
Self-Labeling	0.50019	0.00037	0.00791	0.00811
k-means	0.50179	0.00358	0.00391	0.00432
mb k-means	0.5004	0.00079	0.00282	0.00315

Table 6.4: Metric evaluation 256 cluster

We can see from the results, that for all methods and metrics, the obtained values are not good. All of them have a Rand score around 0.5 and a score close to zero for the other metrics. For comparison, the results for randomly arranged labels are portrayed in Table 6.5.

	Rand Index	adjusted RI	adjusted MI	V-measure
Random	0.5	0.0	0.0	0.0

Table 6.5: randomly shuffled labels

This leads to the conclusion, that comparing the obtained clusters to the labels with the proposed metrics does not reveal any useful information for the quality of the clusters. We can interpret them either as all being randomly grouped together or all clusterings being bad compared to the ground truth. If the assignments are randomly distributed, we would expect the accuracy of our label-mapped models to be around 50 percent, which conflicts with the results of our previous experiments. We think it is more reasonable to say that the used metrics are not very applicable to our dataset, as we only have two classes. For example, the Rand Index takes into account how many examples are not grouped together and are also not in the same ground truth class, so we expect higher values for more ground truth classes.

We can make this more clearly by examining the following example, using the Rand index for comparison:
Suppose we have true labels $[0, 0, 0, 1, 1, 1, 1, 0]$, which are then grouped by a clustering algorithm into two cluster $[0, 0, 0, 1]$ and $[1, 1, 1, 0]$, resulting in the new labeling $[0, 0, 0, 0, 1, 1, 1, 1]$. If we set the first cluster as being class zero and the second one as being class one, we get an accuracy of 75 Percent for each cluster.

The resulting score of the Rand Index is 0.57143.
Now we take the true lables $[0, 0, 0, 1, 1, 1, 1, 0, 2, 2, 2, 3, 3, 3, 3, 2]$ and the clustering $[0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3]$ which results in the same accuracy of 75 Percent as we name the cluster after their most common example. This time we get a much higher Rand Index of 0.8.

In the next step, we compare the found clusters of the methods to each other. From our previous observations, we expect them to give more insight into the similarity of the clusterings, at least for higher numbers of clusters. Results can be seen in Table 6.6 for 2 clusters and in Table 6.7 for 256 clusters. The comparisons of 16 and 32 clusters can be found in the appendix, B.1 and B.2 We abbreviate k-means as km and mini-batch k-means as mb$k$m in the following Tables.

2 Clusters	RI	adjusted RI	adjusted MI	V-measure
DC vs. SeLa	0.51989	0.03978	0.02948	0.02948
DC vs. ODC	0.52018	0.04036	0.02956	0.02956
SeLa vs ODC	0.52089	0.04177	0.03035	0.03035
mb$k$m vs. DC	0.50597	-0.00146	0.00071	0.00072
mb$k$m vs. SeLa	0.50065	0.0013	0.02752	0.02753
mb$k$m vs. ODC	0.5001	0.00011	0.00037	0.00038
mb$k$m vs. km	0.96728	0.2456	0.18059	0.18062

Table 6.6: Clusterings of the methods compared to each other with 2 clusters

For two clusters, the direct comparisons suggest no real similarities, except for k-means and mini-batch k-means. The higher values of the standard clustering algorithms are due to a clustering, where all except a few hundred examples are assigned to the same cluster. By comparing the resulting clusterings of 256 clusters, we can see higher values for all scores. Even though the adjusted RI is still not very high, the other scores suggest that all methods obtained similar clusterings.

6.4 Supervised Training

Most of the state-of-the-art neural network architectures use at least some kind of supervision. For this reason, we wanted to compare the unsupervised methods to a

256 Clusters	RI	adjusted RI	adjusted MI	V-measure
DC vs. SeLa	0.99228	0.14433	0.58169	0.59531
DC vs. ODC	0.99097	0.14695	0.566	0.57994
SeLa vs ODC	0.99266	0.21404	0.63697	0.64852
mb$k$m vs. DC	0.94394	0.05357	0.51814	0.52879
mb$k$m vs. SeLa	0.94481	0.05028	0.51014	0.52106
mb$k$m vs. ODC	0.94397	0.05851	0.47366	0.48494
mb$k$m vs. km	0.8633	0.21783	0.54729	0.55238

Table 6.7: Clusterings of the methods compared to each other with 256 clusters

supervised approach. Another reason is the state-of-the-art model for gamma detection, which is also fully supervised.

The supervised training was conducted in a similar way to the hyperparameter search. We tested multiple configurations of parameters and used only the mapping dataset, consisting of 80.000 images to train the network and 20.000 images for validation. To get a comparable result to the work of Buschjäger et al. [3], we also trained each model on the gamma wobble set. We split the wobble dataset as in the hyperparameter search, resulting in a training set with 100.000 images and a label mapping set consisting of 80.000 images for label mapping and 20.000 images for validation. For the unsupervised methods, we trained each model with their best respective hyperparameters as found in 6.1. The supervised neural network was also trained with the best hyperparameters from the first run. Additionally, we tested the configurations of Buschjäger et al.
We report the best results for each model in table 6.8.

	Diffuse	Wobble	Parameters		
			learning rate	clusters	blocks
Supervised	**79.194**	**90.721**	0.1	–	4
Deep Cluster	70.167	86.217	0.001	256	2
Online Deep Cluster	69.893	85.538	0.001	16	2
Self-Labeling	69.554	85.666	0.001	32	2

Table 6.8: Supervised vs. Unsupervised

The neural network trained under supervision is the clear winner of this comparison, but the gap between unsupervised and supervised is way smaller on the Wobble dataset. This implies that the Diffuse dataset is somewhat harder to classify. This is not surprising as the Wobble data aims at representing strong gamma sources more often, caused by the mode of operation as described in Chapter 4. Therefore, the models can easier distinguish between a strong gamma signal and the weaker hadron radiation.

6.5 Significance

In our final experiment, we use the previously trained models to conduct the Li & Ma significance test, denoted as $S_{Li\&Ma}$, on the real-world data produced by the FACT telescope. $S_{Li\&Ma}$ is a statistical hypothesis test that is applied to photon counts measured, for example, in the Wobble operation mode of a telescope. As already mentioned in Chapter 4, we first have to assume that only the *On* region observes the suspected gamma source. Then the *Off* regions only measure background radiation. With the photon count of the *On* region being N_{on} and the photon count of the *Off* region being N_{off}, we can write the probable number of photons coming directly from the source as:

$$N_s = N_{on} - \alpha N_{off} \tag{6.12}$$

The scalar α is a scaling factor denoting the ratio of the area of the *On* region to the area of the *Off* region. The significance of detection S is then defined as:

$$\begin{aligned} S(N_{on}, N_{off}) = \sqrt{2}\Bigg\{ & N_{on} \ln\left[\frac{1+\alpha}{\alpha}\left(\frac{N_{on}}{N_{on}+N_{off}}\right)\right] \\ & + N_{off} \ln\left[(1+\alpha)\left(\frac{N_{off}}{N_{on}+N_{off}}\right)\right]\Bigg\}^{1\backslash 2} \end{aligned} \tag{6.13}$$

As $S_{Li\&Ma}$ is a hypothesis test, we can formulate the null hypothesis that the *On* region does not observe a gamma source. The value obtained by S then shows by how many standard deviations σ we can reject our hypothesis. Using this test on a fixed dataset can give us an estimation of the detection efficiency of our model. In gamma astronomy, five or more σ are considered a gamma event. Classifying all our images as gamma events gives us a value of around 15 σ, so we aim for at least 15.

For the $S_{Li\&Ma}$ test, we use the subset of the FACT dataset, which was taken while operating in Wobble mode. For the evaluation, we need to determine the class probability of an event being gamma radiation. Therefore we use our models and

make a forward pass with the dataset to derive our predictions. During the forward pass, we convert the model output into class probabilities by performing the softmax operation. Now we can adjust the threshold for an event being classified as gamma radiation. We test multiple thresholds for each model and display the models with the highest significance. The complete evaluation of varying thresholds can be found in the appendix C.1. For each method, we test both models trained on the FACT dataset, of which one was trained for 50 epochs and one for 5. We also test the best-performing model from the hyperparameter search, trained on the Diffuse dataset. Note that after the unsupervised training on the different datasets, the models have to be label-mapped to the simulation data. Otherwise, we do not know which class corresponds to gamma events. As explained at the beginning of this Chapter, this is achieved by only re-learning the classifier of our models while letting the parameters of the layers for feature extraction unchanged. Additionally, we test two supervised models from Section 6.4, one trained on the Wobble and one trained on the Diffuse dataset. Table 6.9 summarizes our findings.

	Wobble	Diffuse	FACT 5	FACT 50
Supervised	**<u>24.45</u>**	23.87	–	–
Deep Cluster	–	18.70	**19.74**	16.49
Online Deep Cluster	–	18.31	**20.61**	18.54
Self-Labeling	–	**19.11**	17.35	18.43

Table 6.9: $S_{Li\&Ma}$ significance test

The highest significance for each method is marked in bold text, and the highest significance overall is additionally underlined.

Comparing only the unsupervised models, we can see that Online Deep Cluster achieves the highest significance. Deep Cluster shows similar differences to our second experiment in Section 6.2 when only trained for five epochs instead of 50 on the FACT dataset. Both methods perform better when trained on the real-world data. Only the Self-Labeling method achieves its best performance on the simulated data. Overall all methods deliver decent results, but the highest scores were achieved by the models trained under supervision, mirroring our results from Section 6.4.

Figure 6.2: Viridis

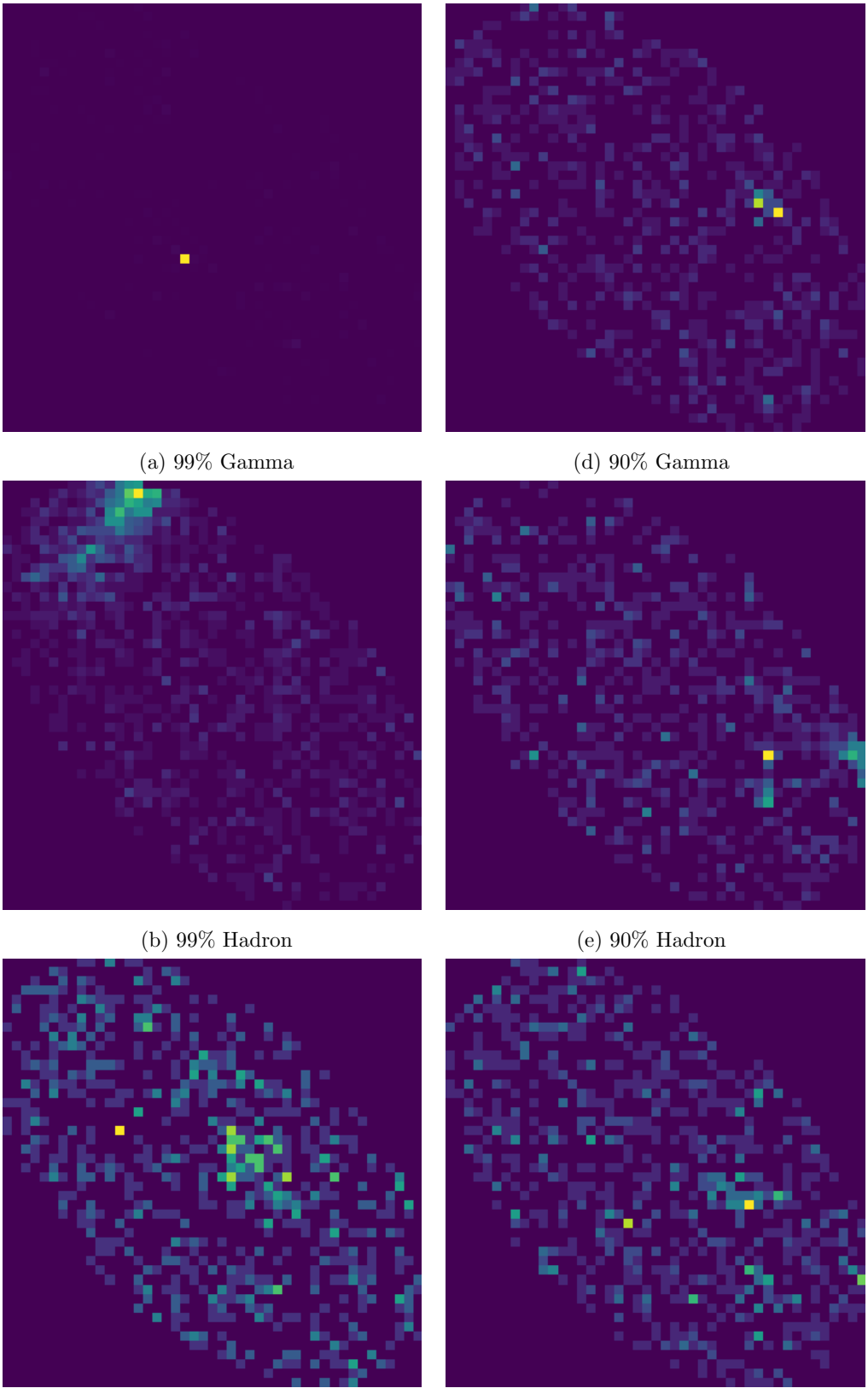
Figure 6.3: BW

6.6 Visualization

Before we close this Chapter and finish this thesis with a conclusion and outlook on possible future work, we present some images identified as gamma radiation by our methods. For each method, we take the best performing model of the $S_{Li\&Ma}$ test and display images with a high probability of being gamma events or background radiation. We create the images by using the Python library Matplotlib [14].

We used the standard coloring scheme of Matplotlib, "Viridis", which has a range from purple for low-intensity pixels to yellow for high-intensity pixels. Figure 6.2 and Figure 6.3 show the range for a ten pixel wide image in comparison to a normal grey scale image. Note that there is not necessarily a one-to-one conversion between each of the ten pixels. The images show only a rough conversion from black and white to the new color spectrum.

We display six images for each method. The first two are identified as gamma events, one with a probability of 99 Percent and one with a probability of 90 Percent. The next two images are identified as background radiation or non-gamma event, again with a probability of 99 and 90 Percent. The last two images have the same probability of being gamma or background radiation. The resulting images can be see in Figure 6.4 for Deep Cluster, in Figure 6.5 for the Self-Labeling method, and in Figure 6.6 for Online Deep Cluster.

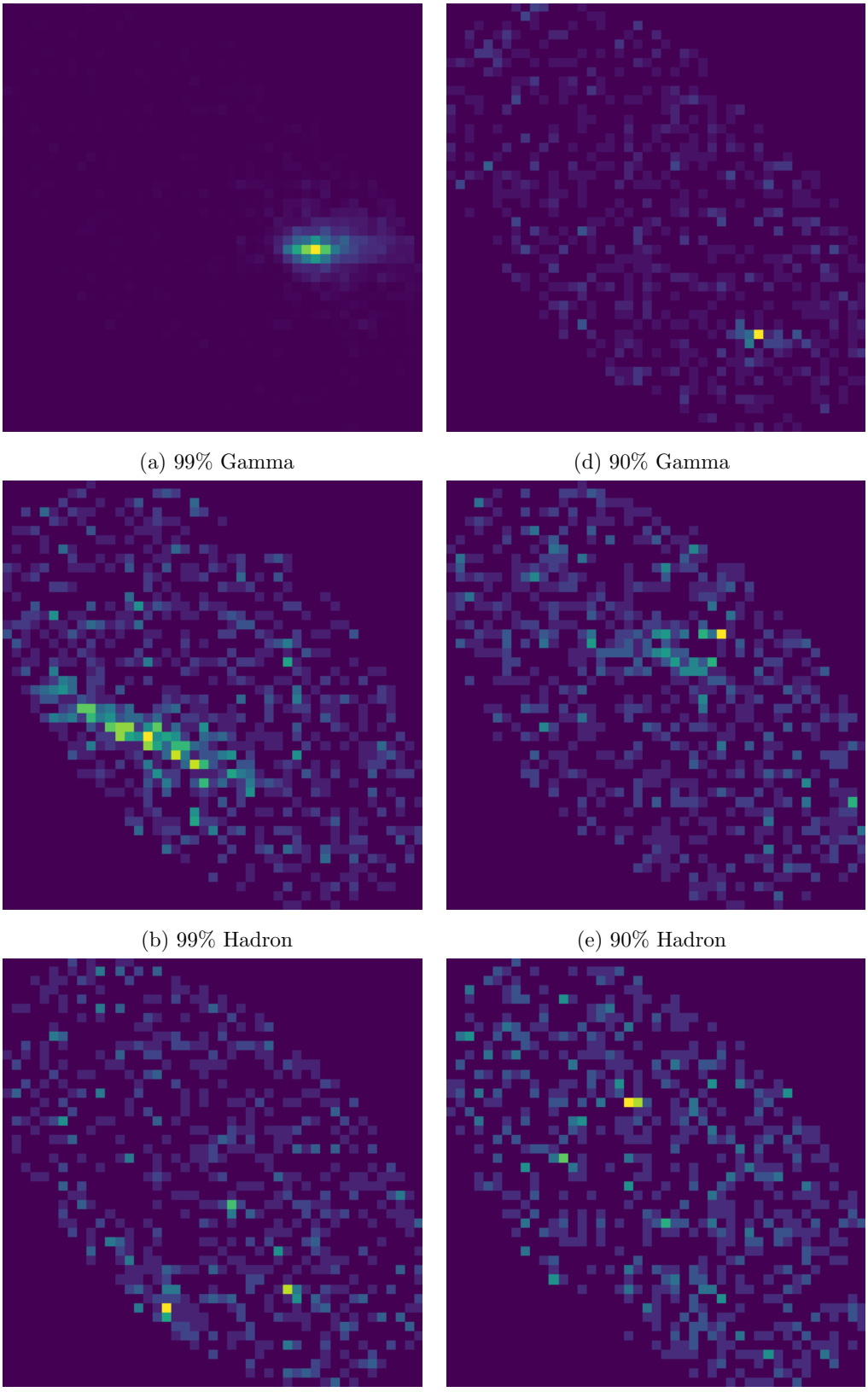


(a) 99% Gamma (d) 90% Gamma

(b) 99% Hadron (e) 90% Hadron

(c) 50% Gamma, 50% Hadron (f) 50% Gamma, 50% Hadron

Figure 6.4: Images classified by Deep Cluster



(a) 99% Gamma

(d) 90% Gamma

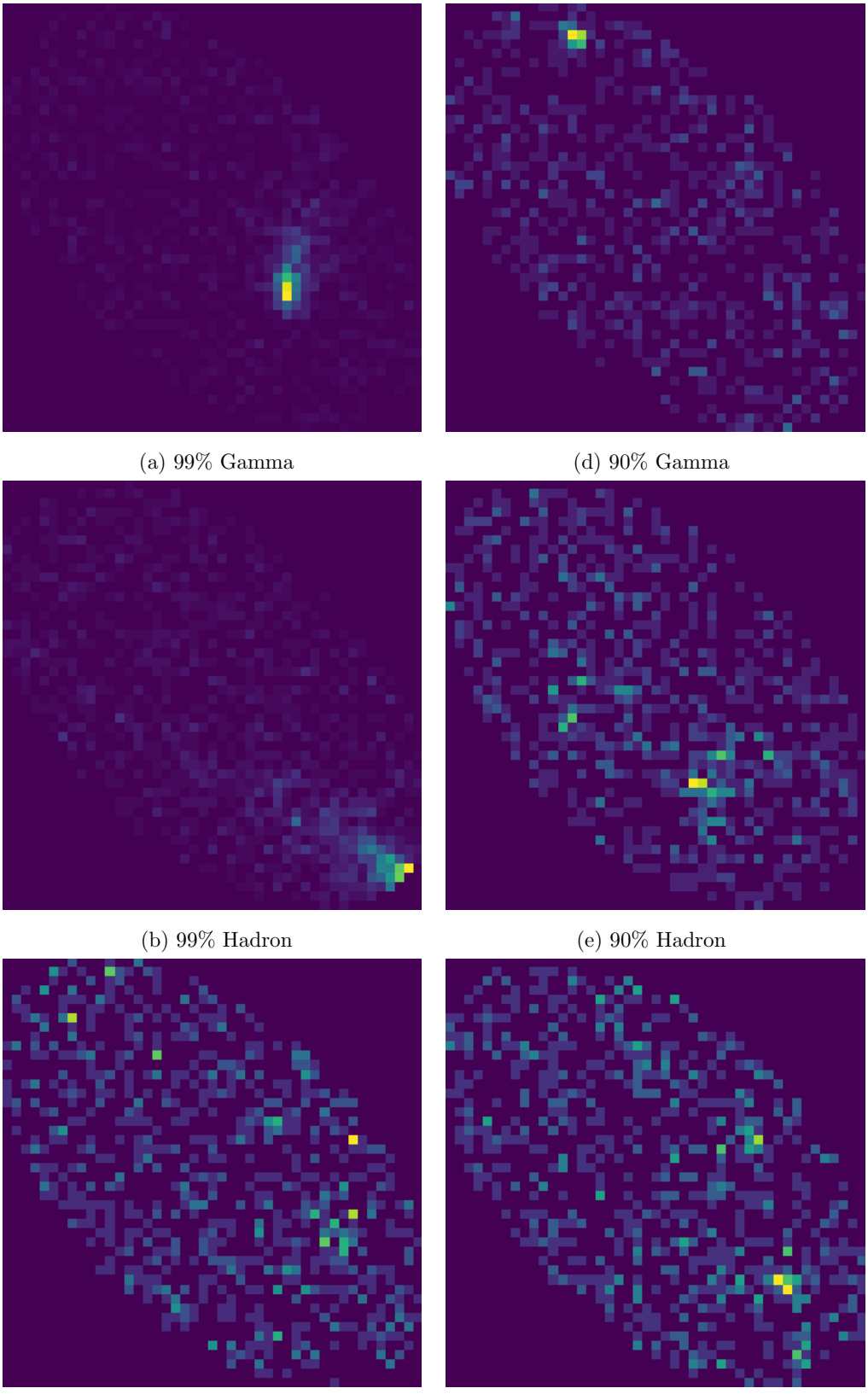
(b) 99% Hadron

(e) 90% Hadron

(c) 50% Gamma, 50% Hadron

(f) 50% Gamma, 50% Hadron

Figure 6.5: Images classified by SeLa



(a) 99% Gamma

(d) 90% Gamma

(b) 99% Hadron

(e) 90% Hadron

(c) 50% Gamma, 50% Hadron

(f) 50% Gamma, 50% Hadron

Figure 6.6: Images classified by Online Deep Cluster

Chapter 7

Conclusion

In this thesis, we presented three recently proposed methods for learning feature representations of a given dataset without the need for supervision. Each method clusters the features extracted by the underlying neural network to derive pseudo-labels for the training process. Several experiments were conducted to test the performance of the methods on the problem of gamma-hadron separation. We started by training our models on the simulated images and the data recorded by the FACT telescope. We then evaluated the trained models in terms of detection accuracy by re-training the classifier to predict whether a given image includes gamma rays or cosmic background radiation. As each method relies on clustering to derive feature representations, we examined the resulting clusters and compared them to a traditional clustering approach, k-means. After comparing our models to the standard supervised approach, traditionally used for neural network training, we concluded our experiments by performing the $S_{Li\&Ma}$ significance test on our label-mapped models.

We showed that all methods produce good results, but can not achieve the level of accuracy of models trained under full supervision. For future work in applying this type of unsupervised clustering approach to gamma astronomy, the $S_{Li\&Ma}$ significance test could be directly integrated into the training process, as accomplished in [21]. After deriving a clustering of the simulation dataset, instead of mapping the classifier onto labeled data, we could aim at maximizing the $S_{Li\&Ma}$ significance on the FACT data by assigning each found cluster to either gamma or hadron radiation.

Appendix A

Hyperparameter

LR	Clusters	DC	ODC	SeLa
0.1	2	64.011	68.759	65.976
0.1	8	66.66	68.224	64.709
0.1	16	65.255	68.705	65.637
0.1	32	67.218	68.599	65.413
0.1	64	66.72	66.931	65.949
0.1	128	67.07	66.885	64.879
0.1	192	61.964	65.694	65.985
0.1	256	65.207	67.381	64.642
0.01	2	60.204	67.668	66.161
0.01	8	67.872	68.387	66.376
0.01	16	66.734	69.374	66.371
0.01	32	65.796	69.59	67.112
0.01	64	64.056	69.706	66.775
0.01	128	64.472	67.784	66.899
0.01	192	61.759	66.933	68.106
0.01	256	64.133	66.622	66.765
0.001	2	63.764	67.379	64.839
0.001	8	66.251	69.645	68.831
0.001	16	66.397	**69.893**	68.319
0.001	32	67.744	69.127	**69.554**
0.001	64	68.54	68.05	69.074
0.001	128	69.905	66.941	69.245
0.001	192	69.54	68.416	69.036
0.001	256	**70.167**	68.114	68.62
0.0001	2	65.956	67.697	64.117
0.0001	8	67.228	66.735	68.269
0.0001	16	67.131	68.893	67.214
0.0001	32	65.706	66.898	69.201
0.0001	64	67.68	66.986	68.144
0.0001	128	66.485	67.631	68.59
0.0001	192	65.995	66.957	69.175
0.0001	256	66.179	66.077	67.534
0.00001	2	66.119	66.045	66.047
0.00001	8	66.169	65.612	66.194
0.00001	16	66.155	65.417	65.932
0.00001	32	64.699	65.353	67.697
0.00001	64	67.07	66.562	65.725
0.00001	128	66.174	65.332	66.091
0.00001	192	65.973	66.063	65.695
0.00001	256	66.203	65.735	65.127

Table A.1: Accurcacy for 2 blocks from the second hyperparameter search as described in Chapter 6.1. The best results from each method are marked in their respective column.

LR	Clusters	DC	ODC	SeLa
0.1	2	66.217	67.923	62.999
0.1	8	65.866	67.671	64.7
0.1	16	66.999	68.257	65.701
0.1	32	66.829	68.422	66.43
0.1	64	66.289	68.278	66.687
0.1	128	65.212	67.853	65.729
0.1	192	66.361	67.689	65.821
0.1	256	60.735	68.07	65.182
0.01	2	58.558	68.025	63.035
0.01	8	67.365	66.857	65.594
0.01	16	67.23	**68.513**	66.083
0.01	32	66.165	67.762	65.967
0.01	64	62.854	68.068	67.837
0.01	128	64.524	66.467	67.597
0.01	192	62.454	66.366	67.78
0.01	256	58.8	65.975	68.877
0.001	2	61.727	66.878	67.037
0.001	8	62.028	68.396	66.473
0.001	16	62.762	66.929	67.83
0.001	32	65.206	66.891	68.515
0.001	64	66.477	67.323	67.372
0.001	128	65.915	67.589	67.993
0.001	192	68.724	66.072	68.949
0.001	256	**69.525**	67.222	**69.517**
0.0001	2	64.816	67.571	67.031
0.0001	8	65.785	66.914	67.597
0.0001	16	66.238	67.996	67.661
0.0001	32	67.101	66.514	68.956
0.0001	64	66.114	66.954	67.886
0.0001	128	65.3	66.998	68.375
0.0001	192	64.949	66.1	66.546
0.0001	256	64.749	66.292	68.243
0.00001	2	64.225	64.834	65.848
0.00001	8	65.51	64.745	64.818
0.00001	16	65.138	65.404	65.548
0.00001	32	65.926	65.383	65.245
0.00001	64	65.492	64.395	64.234
0.00001	128	65.266	64.252	64.378
0.00001	192	64.777	65.176	64.746
0.00001	256	64.689	65.472	65.498

Table A.2: Accurcacy for 3 blocks from the second hyperparameter search as described in Chapter 6.1. The best results from each method are marked in their respective column.

Appendix B

Cluster Comparison

16 Clusters	RI	adjusted RI	adjusted MI	V-measure
DC vs. SeLa	0.86301	0.19609	0.4913	0.49141
DC vs. ODC	0.86704	0.22819	0.56367	0.56377
SeLa vs ODC	0.91925	0.32336	0.54954	0.54963
mb$k$m vs. DC	0.24547	-0.01629	0.06704	0.06739
mb$k$m vs. SeLa	0.22034	0.00246	0.09151	0.09182
mb$k$m vs. ODC	0.22071	0.0009	0.08829	0.08861
mb$k$m vs. km	0.86993	0.40289	0.26044	0.26117

Table B.1: Cluster comparison from Chapter 6.3 for 16 clusters. The obtained clusterings from each model are compared against each other.

32 Clusters	RI	adjusted RI	adjusted MI	V-measure
DC vs. SeLa	0.89983	0.08729	0.41756	0.41799
DC vs. ODC	0.8944	0.07388	0.36961	0.37008
SeLa vs ODC	0.95238	0.2714	0.54816	0.54848
mb$k$m vs. DC	0.31445	-0.05727	0.11291	0.11401
mb$k$m vs. SeLa	0.33667	0.0057	0.15118	0.15213
mb$k$m vs. ODC	0.33655	0.00279	0.12463	0.12564
mb$k$m vs. km	0.78113	0.41046	0.31678	0.31801

Table B.2: Cluster comparison from Chapter 6.3 for 32 clusters. The obtained clusterings from each model are compared against each other.

Appendix C

Significance test

Threshold	Supervised		Deep Cluster			Online Deep Cluster			Self-Labeling		
	Diffuse	Wobble	FACT 50	FACT 5	Diffuse	FACT 50	FACT 5	Diffuse	FACT 50	FACT 5	Diffuse
0.40	19.47	16.16	16.25	19.19	17.71	**18.54**	18.10	16.97	17.81	**17.35**	17.82
0.45	19.84	16.28	16.19	19.71	17.87	17.82	19.03	16.73	**18.43**	16.52	18.01
0.50	20.53	16.54	**16.49**	**19.74**	18.22	18.38	19.80	17.18	18.36	15.97	18.44
0.55	21.52	16.71	15.71	19.54	18.39	**18.54**	19.93	17.28	18.14	15.45	18.55
0.60	22.70	17.34	16.43	19.24	17.72	18.18	**20.61**	17.95	18.34	14.87	18.67
0.65	23.54	17.72	14.65	17.80	17.49	18.42	20.57	17.88	18.20	14.05	18.61
0.70	**23.87**	18.66	15.26	16.94	18.31	17.98	20.23	17.65	17.55	12.48	18.89
0.75	23.58	19.28	14.60	15.45	18.06	16.86	20.28	**18.31**	16.91	10.69	**19.11**
0.80	23.73	19.65	11.90	14.25	18.34	15.02	20.37	18.10	15.84	8.02	18.73
0.85	23.38	20.44	9.53	11.41	18.59	13.40	19.07	17.68	14.90	6.30	18.61
0.90	23.44	20.74	6.86	7.93	18.69	10.26	16.49	17.43	12.91	4.26	17.68
0.95	**23.87**	22.33	3.15	5.95	**18.70**	5.36	11.13	16.30	10.85	2.42	15.86
0.99	17.06	**24.45**	0.0	0.0	12.86	1.62	3.42	9.63	4.72	1.23	9.54

Table C.1: The full $S_{Li\&Ma}$ significance test for each model, including the threshold variations as described in Chapter 6.5. The Table shows multiple trained models for each method. The unsupervised models are trained on the FACT dataset, for 50 and 5 epochs and on the Diffuse simulation dataset for 50 epochs. Each trained model is label mapped on the Diffuse data according to Chapter 6.1. As for the neural networks trained under supervision, one model is trained and label mapped on the Diffuse simulation data and one on the Wobble data.

Bibliography

[1] H Anderhub, M Backes, A Biland, V Boccone, I Braun, T Bretz, J Buß, F Cadoux, V Commichau, L Djambazov, D Dorner, S Einecke, D Eisenacher, A Gendotti, O Grimm, H von Gunten, C Haller, D Hildebrand, U Horisberger, B Huber, K S Kim, M L Knoetig, J H Köhne, T Krähenbühl, B Krumm, M Lee, E Lorenz, W Lustermann, E Lyard, K Mannheim, M Meharga, K Meier, T Montaruli, D Neise, F Nessi-Tedaldi, A K Overkemping, A Paravac, F Pauss, D Renker, W Rhode, M Ribordy, U Röser, J P Stucki, J Schneider, T Steinbring, F Temme, J Thaele, S Tobler, G Viertel, P Vogler, R Walter, K Warda, Q Weitzel, and M Zänglein. Design and operation of FACT – the first g-APD cherenkov telescope. *Journal of Instrumentation*, 8(06):P06008–P06008, jun 2013. https://doi.org/10.1088/1748-0221/8/06/p06008.

[2] YM Asano, C Rupprecht, and A Vedaldi. Self-labelling via simultaneous clustering and representation learning. In *International Conference on Learning Representations*, 2019. http://arxiv.org/abs/1911.05371.

[3] Sebastian Buschjäger, Lukas Pfahler, Jens Buss, Katharina Morik, and Wolfgang Rhode. On-site gamma-hadron separation with deep learning on fpgas. In Yuxiao Dong, Dunja Mladenić, and Craig Saunders, editors, *Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track*, pages 478–493, Cham, 2021. Springer International Publishing. https://doi.org/10.1007/978-3-030-67667-4_29.

[4] Mathilde Caron, Piotr Bojanowski, Armand Joulin, and Matthijs Douze. Deep clustering for unsupervised learning of visual features. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision – ECCV 2018*, pages 139–156, Cham, 2018. Springer International Publishing. https://doi.org/10.1007/978-3-030-01264-9_9.

[5] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, USA, 2006.

[6] cta observatory. Observing the highest-energy processes in the universe. https://www.cta-observatory.org/science/gamma-rays-cosmic-sources/. last accessed: 22-08-2021.

[7] Marco Cuturi. Sinkhorn distances: Lightspeed computation of optimal transport. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, page 2292–2300, Red Hook, NY, USA, 2013. Curran Associates Inc.

[8] J. Knapp F. Schmidt. Corsika shower images. https://www-zeuthen.desy.de/~jknapp/fs/showerimages.html, 2005. last accessed: 22-08-2021.

[9] V.P. Fomin, A.A. Stepanian, R.C. Lamb, D.A. Lewis, M. Punch, and T.C. Weekes. New methods of atmospheric cherenkov imaging for gamma-ray astronomy. i. the false source method. *Astroparticle Physics*, 2(2):137–150, 1994. https://doi.org/10.1016/0927-6505(94)90036-1.

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

[11] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. https://doi.org/10.1038/s41586-020-2649-2.

[12] D. Heck, J. Knapp, J. N. Capdevielle, G. Schatz, and T. Thouw. *CORSIKA: a Monte Carlo code to simulate extensive air showers.* 1998.

[13] Arabie P. Hubert L. Comparing partitions. *Journal of Classification*, 2:193–218, 1985. https://doi.org/10.1007/BF01908075.

[14] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. https://doi.org/10.1109/MCSE.2007.55.

[15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 448–456. JMLR.org, 2015.

[16] Ti-Pei Li and YuQian Ma. Analysis methods for results in gamma-ray astronomy. *The Astrophysical Journal*, 272:317–324, 08 1983. https://doi.org/10.1086/161295.

[17] Tom M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[18] Kevin P. Murphy. *Machine Learning - A Probabilistic Perspective*. MIT Press, 2012.

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[21] Lukas Pfahler, Mirko Bunse, and Katharina Morik. Noisy labels for weakly supervised gamma hadron classification. 2021.

[22] William M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850, 1971. https://doi.org/10.1080/01621459.1971.10482356.

[23] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 410–420, Prague, Czech Republic, June 2007. Association for Computational Linguistics.

[24] D. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. https://doi.org/10.1038/323533a0.

[25] D. Sculley. Web-scale k-means clustering. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 1177–1178, New York, NY, USA, 2010. Association for Computing Machinery. https://doi.org/10.1145/1772690.1772862.

[26] Nguyen Xuan Vinh, Julien Epps, and James Bailey. Information theoretic measures for clusterings comparison: Is a correction for chance necessary? In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 1073–1080, New York, NY, USA, 2009. Association for Computing Machinery. https://doi.org/10.1145/1553374.1553511.

[27] Xiaohang Zhan, Jiahao Xie, Ziwei Liu, Yew-Soon Ong, and Chen Change Loy. Online deep clustering for unsupervised representation learning. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6687–6696, 2020. https://doi.org/10.1109/CVPR42600.2020.00672.