technische universität
dortmund

Bachelorarbeit

Deep Unsupervised Domain Adaptation for Gamma-Hadron Separation

Robin Dylan Drew
März 2021

Gutachter:
Prof. Dr. Katharina Morik
Mirko Bunse

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

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Structure . . . 2

2 Foundations **3**
2.1 Domain Adaptation . . . 3
2.2 Neural Networks . . . 4
2.2.1 Backpropagation . . . 5
2.2.2 Convolutional Neural Network . . . 9
2.3 Classification . . . 12
2.3.1 Logistic Regression . . . 13
2.3.2 Random Forest . . . 13
2.4 FACT / Gamma-Hadron Separation . . . 14

3 Methods **17**
3.1 Deep Unsupervised Domain Adaptation . . . 17
3.2 Baselines . . . 19
3.2.1 Nearest-Neigbor Importance Weighting . . . 20
3.2.2 Logistic Discrimination . . . 21
3.2.3 Transfer Component Analysis . . . 22

4 Experiments **23**
4.1 Setup . . . 23
4.1.1 Cross-Validation . . . 23
4.1.2 Accuracy . . . 24
4.1.3 Performance Metric . . . 25
4.2 Deep Unsupervised Domain Adaptation . . . 26
4.3 Baselines . . . 28
4.4 Implementation . . . 28
4.5 Results . . . 30
4.5.1 Baselines . . . 30

4.5.2 Deep Unsupervised Domain Adaptation 32
4.5.3 Comparison 33

5 Summary and Outlook **35**

List of Figures **37**

List of Tables **39**

References **41**

Erklärung **47**

Chapter 1

Introduction

1.1 Motivation

Machine learning is playing an ever growing role, not just in computer science, but in countless real-world applications. Yet traditional machine learning methods have certain disadvantages: firstly, they must be trained on a large amount of data in order to achieve the wanted performance and secondly, it is assumed that the training and target data are distributed independently and identically.

While in most applications this is not too big of a problem, in many real-world applications there might be few labeled data available for training and it can be very expensive or very difficult to obtain the needed amount of training data. Or the training and target data, even though they might be related, could differ in distribution. Therefore it would be ideal, if there existed a way to eliminate the need of large training datasets or a way to maintain good results for a slightly different, but related target domain. A way to exploit the knowledge from a certain source domain and using it to learn a new task in a related target domain more effectively. Exactly this is the goal of *(unsupervised) domain adaptation*.

Domain adaption methods attempt to learn a mapping between two related domains (the source and the target domain) [1]. In this setting, it is not assumed that the training and target data have an identical distribution. On the contrary, domain adaptation is especially designed for cases in which the source and target domains differ. In the specific case of *unsupervised* domain adaptation, it is also assumed that there is now labeled data available in the target domain.

Therefore, unsupervised domain adaptation is a promising machine learning method for many real-world applications, especially in the field of physics. While physics use cases are ones where there is a vast amount of data, the labeled training data is mostly obtained from simulations, while the recorded deployment data (gathered from real-world experiments/observations) is mostly unlabeled. Simulation data might be vastly available

and similar to real data, however its distribution is slightly different than that of real data. To combat this, domain adaptation attempts to find and focus on representations that have the same distribution in both the source and target domain [1].

One specific application, in which domain adaptation could possibly be quite useful, is that of *Gamma-Hadron Separation*, as most machine learning models attempt to solve this problem by using only synthetic data, because it is difficult to gather labeled data for this task [2].

Gamma rays are emitted by different celestial object and can therefore provide interesting information. However, the telescopes build to capture these gamma rays also, inevitably, capture a lot of background noise, known as hadrons. Therefore, in order to analyze the gamma rays, one must separate the gammas from the hadrons.

Recently, Buschjäger et al. [2] presented good results for the gamma-hadron separation task with a deep machine learning model trained on simulation data, although they did not consider the differences between the two domains (simulation data and real-world data). Hence it would be interesting to see wether a deep domain adaptation method could improve their model for more accurate predictions on the real test data.

This is the question I will try to answer in this thesis, by using the deep model proposed in [2] and adjusting it according to the work of Ganin et al. in [1].

1.2 Structure

My thesis is divided into 5 different chapters. In the second chapter I will give an overview of the fundamental tools needed for this thesis. After that, in chapter 3, the main method of this thesis will be introduced, together with three different baseline methods. In the fourth chapter, the setup of my experiments, as well as their results, will be presented and discussed. Finally, in chapter 5, I will summarize my findings and give a brief outlook on further research.

Chapter 2

Foundations

2.1 Domain Adaptation

The aim of this paper is to improve gamma-hadron separation by using deep unsupervised domain adaptation. Therefore it is vital to understand domain adaptation.

Domain adaptation can be seen as a type of transfer learning. Transfer Learning aims to achieve higher accuracy on data from a certain target domain by training a model on a related source domain or task [3].

In order to further understand transfer learning and domain adaptation, one must first understand the definition of "domain" and "task" as defined in Pan et al. [3]. A domain has two components: the first is a feature space (the features of the data [4]) and the second is a marginal probability distribution (the distribution of said features within the dataset [4]). A task also consists of two components: a label space (the set of labels in the dataset) and an objective prediction function, which is learned from the training data. The objective prediction function is used to predict a label from the label space for an instance from the feature space.

Let $\mathcal{X} \subseteq \mathbb{R}^d$ denote the feature space and $\mathcal{Y}$ the label space, $\mathbb{P}_{\mathcal{X}} : \mathcal{X} \to [0, 1]$ is the probability distribution for $\mathcal{X}$. For both domains, the predictive function $f : \mathcal{X} \to \mathcal{Y}$ is assumed to be identical.

For two domains to be different, by this definition, they may have a different feature space (e.g. different languages in a document classification setting) or a different marginal probability distribution (e.g. simulation and real data). When two tasks are different, they can differ in the label spaces or the objective predictive function [3].

In our particular setting, where the source data consists of simulation data and the target data of real data, the source and target domains are different in that they have different marginal probability distributions (simulation data and real data), while the source and target task stays the same.

The aim to learn a classifier or predictor for a target domain by using a related but different source domain applies to the problem of domain adaptation (DA).

DA itself can be divided into two cases, the first being that of unsupervised DA. In this case, there is no labeled training data available in the target domain [1]. This is the case that will be the focus of this paper, since the real data of a telescope is never labeled. The second case is semi-supervised domain adaptation, in which there is some labeled training data available.

Some research has already been conducted on "shallow" DA approaches, creating a mapping between the source and target domains and combining the already learned classifier with this mapping in order to also apply it to the target domain [1]. This can be done by, amongst other things, reweighing or selecting samples from the source domain [1, 5, 6, 7] or by mapping the two domains to another feature space where they cannot be distinguished [8]

Ganin et al. [1] propose a "deep" DA approach in which domain adaptation and (deep) feature learning are combined within one process, instead of being done separately. Therefore the final classifier makes decisions based on discriminative features which have very similar distributions in both domains. I will be applying this approach to the CNN-based model used for gamma-hadron separation developed by Buschjäger et al. [2], in the interest of increasing the predictive accuracy in the target domain by making use of the unlabeled real data.

2.2 Neural Networks

An ANN is a layered network of so called artificial neurons (ANs). Usually an ANN has an input layer, an output layer and one or more hidden layers. The ANs in one of these layers can be partially or fully connected to the ANs in the next layer, with each connection having an assigned weight [9, 10].

Each AN computes an output by calculating the weighted sum of its input connections and applying a non-linear activation function to it. The net input signal is computed as the weighted sum of all input signals and a bias weight. The computed output is then fed to the connected ANs in the next layer [9].

Before training the ANN, the weights of all connections are randomly initialized. Then during the training, the weights are adjusted in order to minimize the error between the networks output and the training datas output [9]. As a neural network's predictive accuracy relies heavily on its training, it is of utmost importance to understand this training procedure. In the learning phase, the aim is to find the best parameters for a specified network so that its predicted outputs/labels $\hat{y}$ are close to the actual outputs/labels y [11], while still being able to adapt to "unseen" data (test data). The most common approach

to training neural networks is that of stochastic gradient descent (SGD) with the help of backpropogation.

2.2.1 Backpropagation

Backpropagation actually consists of two phases: the forward propagation and the backward propagation [12]. The forward propagation serves the purpose of calculating an output for a specific input, while the backward propagation aims to compute the gradient of the networks parameters based on the previously calculated output. But before I go into further explanation about the backward propagation, let me first clarify how the output is calculated during the forward propagation phase. As stated above, the artificial neurons in an ANN compute the net input signal and use this to determine an output according to its activation function. The net input signal is calculated as follows:

$$net_{xj} = \sum w_{ji} o_{xi} \quad + \; b_j \; . \tag{1}$$

Here, net_{xj} is net input signal of the jth neuron given the input x, o_{xi} is the output of the ith neuron and w_{ji} is the weight of the connection between the ith and jth neuron. b_j denotes the aforementioned bias weight of this neuron. Once the net input signal has been calculated, it is passed to the activation function for the determination of the output signal. The most used activation functions are linear and sigmoid functions. While the linear function is pretty self-explanatory, the sigmoid function can be implemented in several way. However,

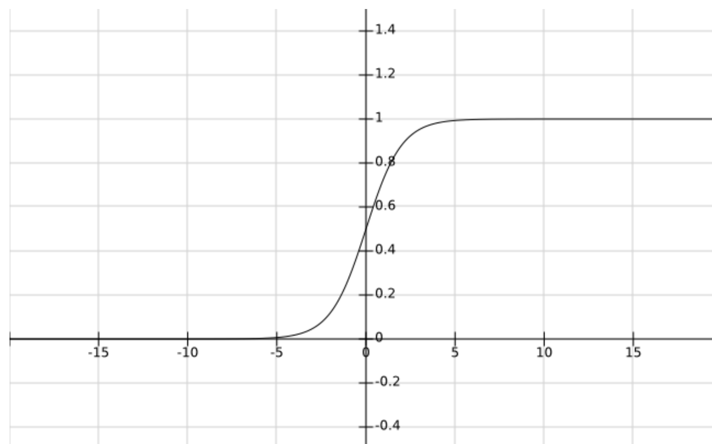
$$o_{xj} = a(net_{xj}) = \frac{1}{1 + e^{-net_{xj}}} \; , \tag{2}$$

with o_{xj} being the output of the jth neuron for the input x, a denoting the activation function and net_{xj} its previously computed net input signal, is the most often used implementation [12, 11, 13]. A sigmoid function only returns values in the range [0,1] as can be seen in Fig. 2.1(a).

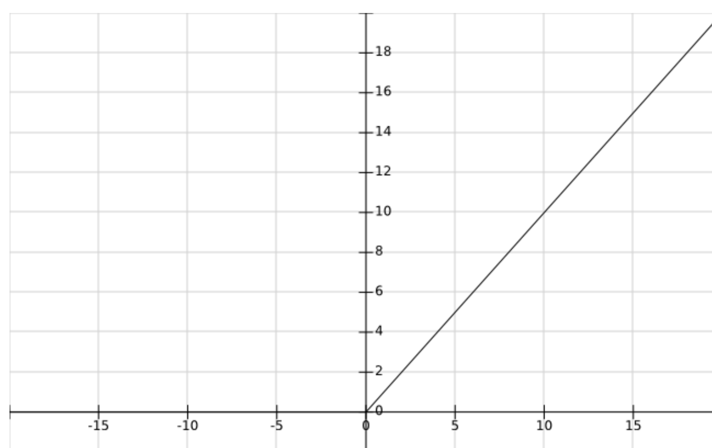
Another very popular activation function is ReLU (Rectified Linear Unit) as seen in Fig. 2.1(b), which has gained popularity by being very light, effective and reliable [14]. ReLU is defined as:

$$o_{xj} = a(net_{xj}) = \max(0, \; net_{xj}) \; , \tag{3}$$

Some of the benefits of the ReLU function are an improved speed in training and fewer activations within the network as only neurons with a positive *net* are activated [15].



(a) Sigmoid.



(b) ReLU

Figure 2.1: (a) illustrates the basic sigmoid for a neuron with a single input. In this case the weight of the input was fixed at 1 and the bias was set to 0. Changing the bias would result in a shifting of the sigmoid along the x axis, while a change in the weight would affect the steepness of the function [12]. (b) depicts the popular ReLU activation function. The functions were plotted using FooPlot[1].

After the net input signal and activation function have been computed in a neuron, the output is fed to all the connected neurons in the next layer. These neurons then do exactly the same until the output layer is reached. Here, instead of feeding the output to the next layer, the output vector is interpreted as the networks output $\hat{y}$ for the given input x. Once the output has been determined, it is compared to the actual desired output y of the input x via some loss function [12].

[1] http://www.fooplot.com

This moves us on to the backward propagation, where the gradient of the networks parameters are computed according to the results of the loss function and then adjusted via stochastic gradient descent. Two popular loss functions are the mean squared error (MSE) [12] and the cross-entropy loss functions [16]:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \ , \tag{4}$$

$$CE = -\sum_{i=1}^{n} y_i \log \hat{y}_i \ , \tag{5}$$

with y_i being the desired output for the i-th training sample, $\hat{y}_i$ the computed output for the i-th training sample and n the number of training samples. If $n = 1$, the loss is calculated after each training sample. This is called *incremental learning*. If $n > 1$, the loss is summed up over samples before any adjustments to the networks parameters are made. This is called *batch learning* [12].

There are reasons for the use of each of the afore mentioned loss functions, however MSE is mostly used for regression tasks, while cross-entropy is used primarily for classification tasks [17].

Now the aim of a neural network is to make predictions $\hat{y}$ that are close to the desired values y. Therefore it must try to minimize the loss function. This can be done by adjusting the weights and biases within the network. But this raises the question of how to adjust them. In order to answer this question we make use of the derivatives of the loss function [11], with respect to the networks parameters, as it tells us wether increasing a weight leads to more or less error. If it leads to more error, the weight is decreased, if it leads to less error, the weight is increased. After the parameters have been adjusted, we go back to the forward propagation. This process is then repeated until the error has reached a certain value or until it has settled down.

So what the network aims to find is a gradient that determines by how much each weight or bias is adjusted. In order to compute such a gradient, one must first compute the derivative of the loss function with respect to each weight and bias:

$$\frac{\partial L_x}{\partial w_{ji}} \ , \ \frac{\partial L_x}{\partial b_i} \ . \tag{6}$$

Here, L is the loss function and w_{ji} and b_i are certain weights and biases. Once the derivatives and the gradient have been determined, the weights and biases can be updated by SGD. To compute the gradients, backpropagation uses something called the chain rule, which makes it possible to calculate all derivatives with only one pass through the network [11].

This method was first proposed by Rumelhart et al. in 1985 [18]. By applying the chain rule, you get following equations:

$$\frac{\partial L_x}{\partial w_{ji}} = \frac{\partial L_x}{\partial net_{xj}} \frac{\partial net_{xj}}{\partial w_{ji}}. \tag{7}$$

In this equation, L_x denotes the calculated loss for input x, w_{ji} is the weight of the connection between the jth and ith unit. net_{xj} is the net input signal of the jth unit given the input x. The first part of this equation shows how the loss changes when the net input signal changes and the second part shows how the net input signal changes when the weight w_{ji} changes. By Eq. 1, the second part of Eq. 7 is:

$$\frac{\partial net_{xj}}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \sum_k w_{jk} o_{xk} + b_k = o_{xi}. \tag{8}$$

For the first part of Eq. 7 we must once again apply the chain rule:

$$\frac{\partial o_{xj}}{net_{xj}} = \frac{\partial L_x}{\partial o_{xj}} \frac{\partial o_{xj}}{net_{xj}}. \tag{9}$$

By Eq. 2 and Eq. 3, the second part of Eq. 9 is:

$$\frac{\partial o_{xj}}{net_{xj}} = a'(net_{xj}), \tag{10}$$

with a' being the derivative of the activation function. For the first part of Eq. 9, we must differentiate between two cases: Firstly, that unit u_j is an output unit and secondly, that it is not an output unit. If u_j is in fact an output unit, then

$$\frac{\partial L_x}{\partial o_{xj}} = \frac{\partial}{\partial o_{xj}} L_x, \tag{11}$$

which is the derivative of the loss function for the output of unit u_j. For the case that unit u_j is not an output unit, we can use the chain rule once again:

$$\frac{\partial L_x}{\partial o_{xj}} = \sum_k \frac{\partial L_x}{\partial net_{xk}} \frac{\partial net_{xk}}{\partial o_{xj}} = \sum_k \frac{\partial L_x}{\partial net_{xk}} \frac{\partial}{\partial o_{xj}} \sum_l w_{kl} o_{xl} + b_k = \sum_k \frac{\partial L_x}{\partial net_{xk}} w_{kj}. \tag{12}$$

This can be easily computed, since all units u_k lie in the layer after u_j and therefore all $\frac{\partial L_x}{\partial net_{xk}}$ have already been computed.

This same method of applying the chain rule can be done to compute the derivative with respect to the bias $\frac{\partial L_x}{\partial b_{ij}}$

Once the derivatives for each weight and bias have been calculated by backpropagation, the gradients are used to adjust the parameters via stochastic gradient descent (SGD). Since the gradient shows how the loss is changed with each parameter change, SGD adjusts the parameters into the steepest decreasing direction and hence reducing the loss.

2.2.2 Convolutional Neural Network

One specific variation of ANNs that is used in our and many other real-world applications are so called *convolutional neural networks* (CNNs) which are designed to work with two-dimensional input data. Therefore they are mostly used for tasks related to some form of pattern recognition in images [19].

A basic CNN can be broken down into two main parts: the feature extractor and the label/class predictor. While the label predictor, often referred to as the fully-connected layer, is a normal, fully-connected artificial neural network, the feature extractor consists of special layers, the so called convolution and pooling layers [19, 20] (see Fig. 2.2). A CNN usually has multiple sets of convolution and pooling layers, with the first layers extracting *low-level features* such as edges and corners, and the later layers finding more precise features (*mid-level* and *high-level features*) such as objects (e.g. eyes, wheels, etc.) [21]. Before the introduction of CNNs, pattern recognition could only be done by feeding handcrafted features to an ANN [22]. With the help of convolution and pooling layers, this can be done automatically.

So let's have a look at what these "special" layers do.

Convolution Layer

As can be seen in Fig. 2.2, a convolution layer consists of multiple feature maps, where each feature map is the result of some filter being applied to all positions in the input [21, 22, 23]. Therefore each feature map describes the occurrence of a specific feature within the input (e.g. a diagonal line in an image). These filters are smaller in size than the input itself so that features can be found in different locations within the input. The way the feature maps are computed is by first applying the filter to the top-left corner of the (two-dimensional) input and then moving the filter a certain number of steps. This number of steps can be set via a parameter called *stride*. If the stride is set to n, the filter is moved n data points (e.g. pixels) after being applied. This process is repeated until the entire

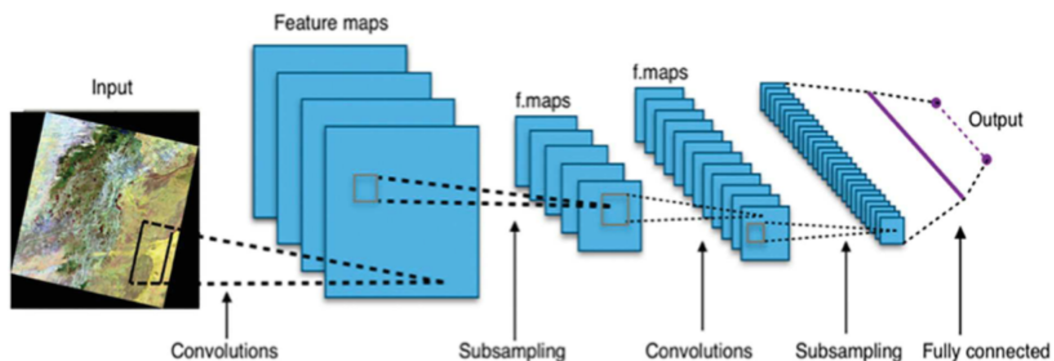
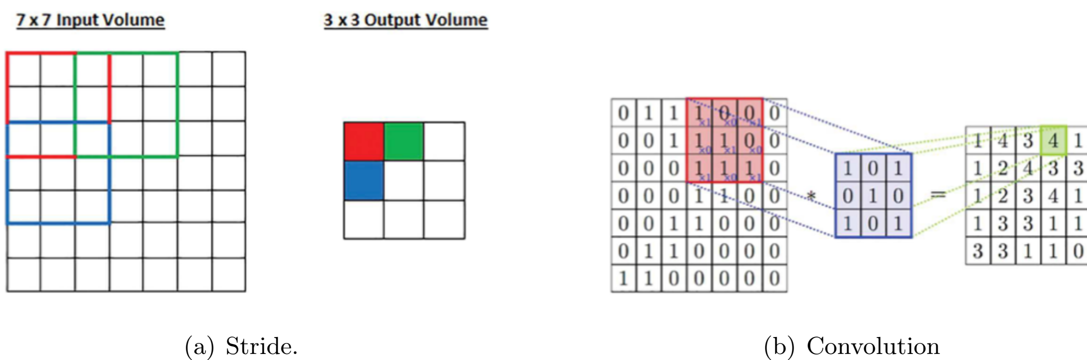


Figure 2.2: A basic representation of a convolutional neural network architecture [22].



(a) Stride. (b) Convolution

Figure 2.3: The convolution process. (a) depicts where the filter is applied to the image with stride 2 and where the results are saved in the feature map. (b) shows how the filter is applied to a certain patch of the image by calculating the dot product [22].

image has been scanned (from left to right and top to bottom). Each time the filter is applied, the dot product of the filter and the input's data points within the frame of the filter is calculated and saved in the feature map for this filter. The convolution process is illustrated in Fig. 2.3. By repeating this process for a number of filters, multiple feature maps are created, each corresponding to a different feature.

As long as we know where each feature is located, relative to other features, their exact locations are not further needed. Thus the feature maps are fed to a pooling layer where they are simplified.

Pooling Layer

The aim of the pooling layer is to simplify the feature maps computed in the convolution layer in order to make the model more efficient and less sensitive to shifts and distortions of the input [21]. Similar to the convolution layers, the pooling layers apply a filter to the feature maps. This filter most commonly returns the maximum value within its frame (this method is called *max pooling*) and is then moved by as many steps as it is long (for a filter of size 2x2, the stride would be 2). This way there is no overlapping and the dimension of the feature maps is noticeably reduced [22, 23]. Another frequently used pooling method is *average pooling*, where instead of returning the maximum value, the filter returns the average of all values within its frame. The more popular *max pooling* is depicted in Fig. 2.4.

In the first convolution layer, the feature maps are computed straight from the input. In the later layers, however, the features are extracted from already extracted features. So instead of looking for specific features in the input data, the later convolution layers aim to find higher level features which can be seen as a combination of low-level features. In order to find such features, the later convolutional layers are connected to the outputs from the previous layers via some weighted connections. These sets of weighted connections is called

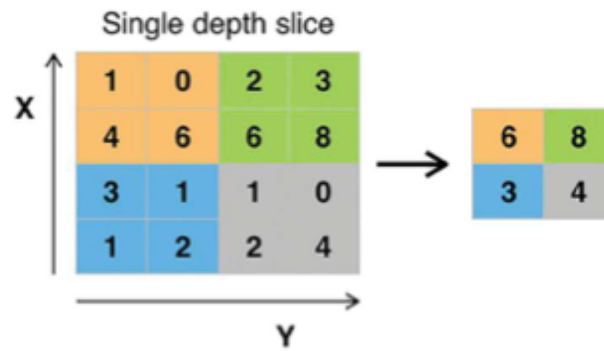


Figure 2.4: The max pooling process. Each colored section depicts a patch covered by the filter. For each patch, the filter returns its maximum value [22].

a *filter bank* [23]. Each feature map within a layer has a different filter bank, as some low-level features might be of greater importance for this feature and some might be irrelevant for this specific feature map. Therefore the feature map is strongly connected to those feature maps providing the desired features. With this concept, the features extracted become more precise, the further into the network we go.

2.3 Classification

Maybe the biggest applications for machine learning models such as convolutional neural networks are classification tasks. Let X be the space of observations with each observation consisting of n features and Y be a set of labels. Then the aim of a classifier is to find a mapping $X \to Y$ or in other words to find a relationship between the features of the observations and their class labels, in order to predict a label $y \in Y$ for an unlabeled observation $x \in X$ based on its features $(x_1, ..., x_n)$.

Some examples of classification tasks are (including some exemplary papers):

- Text classification (i.e. determining the topic of a given text) [24, 25, 26]
- E-mail spam detections (i.e. classifying e-mails into two classes: spam and not spam) [27, 28]
- Image classification (i.e. classifying what can be seen in an image) [29, 30, 31]
- Handwriting recognition [32, 33]
- Speech recognition (i.e. classifying a speakers emotion) [34, 35]
- Music classification (i.e. classifying the musical genre) [36, 37]
- Consumer behavior classification (i.e.) [38, 39]
- Medical classification tasks [40, 41]
- Disease recognition (i.e. detecting diseases based on x-ray images or medical records) [42, 43]

Our problem (gamma-hadron separation) can also be classed as a classification task, because we take the observations from a telescope and decide wether the image displays interesting gamma rays or less interesting hadron noise. Therefore this is a classification task with two possible classes. As stated above, CNNs are machine learning models that can be used as a classifier and they are also the main approach we will be testing in our domain adaptation approach. As our approach can be classed as "deep", we will also be comparing it to some "shallow" domain adaptation approaches which make use of different classifiers. So in order to get a better understanding of theses other methods, I will also be introducing the classifiers used for these approaches.

2.3.1 Logistic Regression

Logistic regression is used for classification problems with two different outputs (i.e. 0 and 1). It determines the relationship between one or more predictor variables x and a response variable y [44]. Via a logistic function and the maximum likelihood function, logistic regression attempts to estimate the parameter vector β that returns the best possible prediction accuracy [45]. When predicting, the linear regression model outputs continuous values between 0 and 1 (i.e. the probability of the sample being in a certain class). However, for classification a clear output (either 0 or 1) is needed. Thus, a logistic regression classifier has a threshold (usually 0.5) that determines the classification [45]. If the output calculated by the linear regression is greater than this threshold, the classifier's output will be 1 and if it is lesser than the threshold the output will be 0.

Since each response variable y can only be 0 or 1, it follows the Bernoulli Probability density function with π being the probability of y being 1 and $(1 - \pi)$ the probability of y being 0 [45]. Therefore, the predictive function is:

$$p(y = 1) = \pi = \frac{1}{1 + e^{-x\beta}} \ . \tag{13}$$

In order to find the best parameter vector β, logistic regression models are optimized by maximum likelihood [46]. The log-likelihood for n observations is [45]:

$$L(\beta/y) = \sum_{i=1}^{n}(y_i \ln(\pi_i) + (1 - y_i)\ln(1 - \pi_i)). \tag{14}$$

To maximize the log-likelihood, its derivative is set to 0 [46].

2.3.2 Random Forest

One classifier that is very popular in the field of gamma ray classification (gamma-hadron separation) is the so called Random Forest (RF). A RF classifier is built up out of multiple tree classifiers [47], which on their own are relatively weak due to a high variance [48]. Thus RFs aim to make these more stable in an ensemble learning setting (i.e. using multiple classifiers to achieve a greater accuracy and more stable predictions). When making predictions with a RF classifier, the input is fed to all trees within the forest which in turn link the input to some class. The overall output is then determined by selecting the class with the most hits/votes. If not one single class has the most votes but multiple classes share the same amount of votes, then one of theses classes is chosen at random [48].

To get a further understanding of how random forests work, we must first understand how their underlying models, the decision trees, work.

A decision tree can be seen as a classifier which classifies its inputs by successively applying a number of decision functions, with each function narrowing down the possible classes [49]. The main components of a decision tree are its nodes. Each tree has a root

node, some interior nodes and some terminal nodes, also called leaves. While terminal nodes, as the name suggests, contain the classes (one terminal node resembles exactly one class), the root and interior nodes contain the decision functions [49]. At each node, one single attribute of the input data (this is the most common form of decision trees; there are also decision trees where multiple attributes are used in a node [49]) is tested via the decision function of this node and depending on the output of this function, the next node is selected. In order to determine which attribute is tested at which stage of the tree, impurity measures such as information gain, gain ratio or gini index can be used [50]. The implementation used for our experiments uses the gini index. The gini index is a value in the range [0,1] that indicates the impurity of the data. So when applied to a single attribute, it expresses wether a given attribute belongs to only one class (pure - gini index of 0) or wether it is distributed randomly amongst multiple classes (impure - gini index of 1). Given a numerical attribute, a splitting point has to be chosen so that the node splits the data into to groups. The impurity measure can be used to find the best splitting point for a certain feature and also to find the feature with the best splitting point.

Now that we have a basic understanding of decision trees, we can get back to the random forest classifiers. For RFs to be accurate, the trees must be uncorrelated. Therefore we cannot just train/create all trees using the same set of data. Instead there are different methods to combat this problem, the most popular being one proposed by Breiman in 2001 [47]. Breiman's method makes use of bagging ("bootstrap aggregation") [51] and random feature selection. Instead of using the complete training data set for the construction of each tree, a random subset of the complete data set is selected for each tree and the tree is grown on the basis of this subset. Now, we already know how a basic decision tree is constructed, but in random forests the construction differs slightly. Just like with normal decision trees, an appropriate attribute/feature to perform a split over is selected at each node of a tree. But instead of going through every attribute/feature and picking one according to some impurity measure, in RFs a new random subset of the data set used to build this tree (which is already a subset of the original training set) is selected and one of the attributes in this random subset is then selected with the help of some impurity measure.

2.4 FACT / Gamma-Hadron Separation

As stated above, domain adaptation can be of great use for many real-world applications due to the lack of real training data. Therefore DA might also improve predictive accuracy in the field of astronomy, more precisely in the *gamma-hadron separation problem*.

In order to further study the universe, telescopes are used to observe high energy beams to identify celestial objects such as supernovae [2]. The interesting part of these high energy beams are the gamma rays, which are surrounded by a lot of background noise, caused by

hadronic cosmic rays. Therefore it is necessary to find a way of separating the gamma rays from the background noise, also called the *gamma-hadron separation problem*.

Most current approaches aim to extract features from the observed data based on rules hand-crafted by experts [Bockermann et al.]. Since these rules are often influenced by some assumption made about the data, the raw data might not support these assumptions, but the ML classifier might [2]. Therefore the work of [1] might be helpful in finding more discriminative features.

Buschjäger et al. created a deep learning model for more precise on-site gamma-hadron separation in the context of the First G-APD Cherenkov Telescope (FACT). The FACT telescope detects Cherenkov light, which is emitted by air showers created by the interaction of particles (e.g. cosmic ray beams) with the earth's atmosphere [2] (see Fig. 2.5). In order to train their model, Buschjäger et al. used the particle simulation software CORSIKA [52] to simulate air showers. These simulated air showers were then run through a simulation of the FACT as to produce realistic data. CORSIKA produced 200k training examples and 100k test examples in two different variants. One with quality cuts (elimination of unrealistic simulated events) and one without [2].

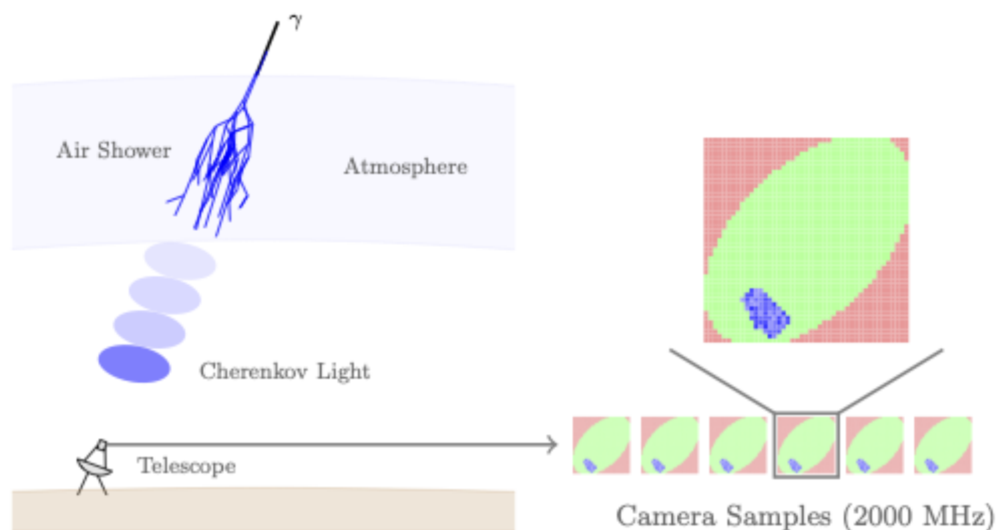


Figure 2.5: An illustration of the emergence of Cherenkov light that is in turn detected by the telescope (left) and a camera sample from said telescope (right). [2].

Chapter 3

Methods

Many methods for gamma-hadron separation have been proposed, but most of them rely on simulated data. While simulated data is very close to real data, there are still differences, mainly in the probabilistic distribution. Buschjäger et al. [2] created a deep learning model which was trained purely on simulated data and still achieved good accuracies on real data. Therefore one must ask the question wether the accuracy of gamma-hadron separation can be further improved by the incorporation of real data into the training process. To answer this question, I will examine a "deep" domain adaptation approach based on the work of Ganin et al. [1] and Buschjäger et al.. I will also test a selection of "shallow" approaches which will pose as a baseline against which the DA approach can be compared.

In the next few sections I will elucidate each of the methods I tested plus the one that is the focus of this thesis. How these methods were implemented, which alterations were made and how they performed will be outlined in section 4.

3.1 Deep Unsupervised Domain Adaptation

The goal of Ganin et al. [1] approach is to accurately predict a label y for a certain input x. At training time they have access to labeled source domain data and unlabeled target domain data. In an effort to distinguish between source and target data, they introduce a binary variable d_i (*domain label*). If the input x_i comes from the source domain, d_i is set to 0, and if it comes from the target domain it is set to 1.

Ganin et al. then propose a deep feed-forward architecture (Fig. 3.1) that not only predicts the label y for each input x but also its domain label d. Therefore the model consists of three different parts. The first part is a *feature extractor* which maps the input x to a feature vector $\mathbf{f} \in \mathbb{R}^D$. The second part is a *label predictor*. Here the feature vector $\mathbf{f}$ is mapped to a label y. These first two parts resemble a standard feed-forward architecture or in our case a standard CNN, similar to the one proposed by Buschjäger et al. [2]. The third part, the *domain classifier*, is what realises the (unsupervised) domain adaptation.

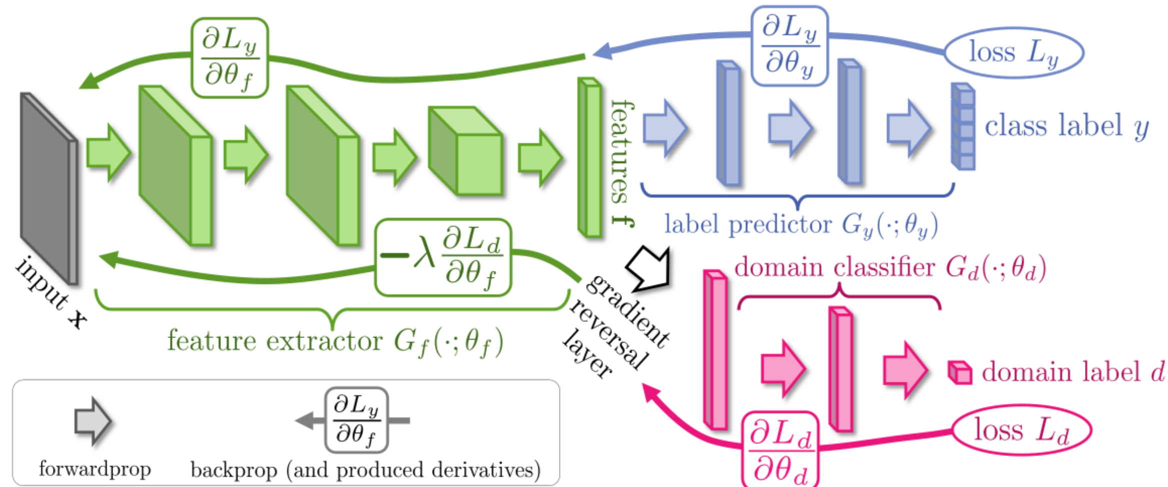


Figure 3.1: The proposed CNN architecture for deep unsupervised domain adaptation by Ganin et al. [1].

In this part, which is connected to the feature extractor via a *gradient reversal layer*, the feature vector is mapped to the domain label d.

As stated above, Ganin et al. aim to find discriminative features which are domain-invariant. The discriminativeness of the features is achieved by optimizing the parameters of the feature extractor and label predictor and thus minimizing the label prediction loss (for source data). The domain-invariance is ensured by the domain classifier and the gradient reversal layer. The dissimilarity of the source and target domain features is estimated by the domain classifier loss L_d (the higher the loss, the greater is the similarity between the source and target domain features). Therefore, the parameters of the feature extractor are optimized in a way that maximizes the domain classifier loss, while the domain classifier parameters are optimized to minimize the domain classifier loss. This means that the more the domain classifier is improved, the more indistinguishable the features and vice versa. This results in a label predictor that focuses on domain-invariant features while paying less attention to features that are dissimilar across the source and target domain.

The gradient reversal layer (GRL) is in turn used to enable optimal label prediction and also domain-invariant features [53]. During the foreword propagation, the GRL performs no permutation to the data and acts as a simple connection between the feature extractor and the domain classifier. During the backpropagation-based training, however, the GRL multiplies the gradient by $-\lambda$, resulting in a reversed gradient. This reversed gradient ensures that the feature extractor pics out domain-invariant features, as it aims to maximize the domain classifier loss L_d. Since stochastic gradient descent adjusts the weights and biases by going into the steepest decreasing direction, via the reversed gradient, the adjustments are made into the steepest increasing direction of L_d. At the early stages of training, this gradient reversal can be disruptive to the improvement of the label predictor in light of the fact that the domain classifier loss is large due to insufficient training [53].

Therefore, λ is gradually changed from 0 to 1 based on the number of network updates p using the following equation:

$$\lambda = \frac{2}{1 + e^{-\gamma p}} - 1 \tag{15}$$

In this equation, γ is a parameter that controls how fast λ reaches the value 1. Further information on this can be found in section 4.2.

In our case, the source data will be the same as it was in Buschjäger et al. (simulated telescope data using CORSIKA). The target data will be real telescope observations. To apply the method developed by Ganin et al., I will extend the CNN used by Buschjäger et al.. Their model consists of four blocks. Each block consists of two 3x3 convolutional layers each followed by a batch normalization layer, a ReLU activation and a single max-pooling layer of stride 2 [2]. These blocks are then followed by a linear layer of size 512 and a final softmax layer. In order to contain the domain adaptation elements, this model will be extended by connecting a domain classifier to the end of the last block via a gradient reversal layer. Different models will be trained and tested on the target data. The models to be tested are mentioned in section 4.2.

3.2 Baselines

In order to find out how good this method works, one must compare it to over methods as well. Since our main domain adaptation method falls into the category of *deep* domain adaptation, it would be interesting to see how well it performs compared to a *shallow* approach. While I tested a number of "shallow" domain adaptation methods, I will only be focusing on a few of those, namely the best performing, the worst performing and one that I find to be the most intuitive, while also producing very good results. These three methods are:

- Nearest-Neighbor Importance Weighting [54]
- Logistic Discrimination [55]
- Transfer Component Analysis [8]

The first two of these methods fall into the category of *importance weighting* domain adaptation. This means that each source sample is assigned a weight based on some relation to the target data, before being fed to some classifier (e.g. logistic regression or random forest classifier). The third method is slightly different, as it maps the original data to another feature space. But more on that in section 3.2.3.

3.2.1 Nearest-Neigbor Importance Weighting

The Nearest-Neighbor Importance Weighting (NNeW) method, proposed by Loog in [54], aims to weight the source samples based on how close they are to the target samples. Therefore it relies on an importance weighting function $w(x)$ which reweighs the source data so one can estimate the target loss with knowledge about the source domain. With the aim of estimating such a weighting function w, Loog tries to find an interpolation to control which source samples are influenced by which target samples by assigning each source sample an individually estimated weight. In order to find such a weighting function, he utilizes the Voronoi cells V_i containing each source sample x_i [56] (Fig. 3.2).

Within each cell V_i one finds the convex set of every point in the feature space which is closer to x_i than to any other source sample. Loog then suggests to estimate the weights of each source sample x_i with the number of target samples ξ_j within its cell V_i:

$$\hat{w}(x_i) = |V_i \cap \{\xi_j\}_{j=1}^{N_\tau}|. \tag{16}$$

For the purpose of obtaining all $w(x_i)$, a nearest neighbor classifier can be trained with each x_i being its own class. When a target sample ξ_j is classified to a certain x_i, the weight $\hat{w}(x_i)$ is increased by one. After estimating the weights, Loog uses either a linear discriminative analysis (LDA) or a quadratic discriminant analysis (QDA) for learning a predictive function from the weighted source domain data. The implementation I will be using [58], however, instead uses either a logistic regression classifier or a random forest classifier because these methods better fit the telescope data (section 4.3).

In his research, Loog found his NNeW method to be very stable, due to a nearest neighbor search being its most complicated operation, while at the same time displaying

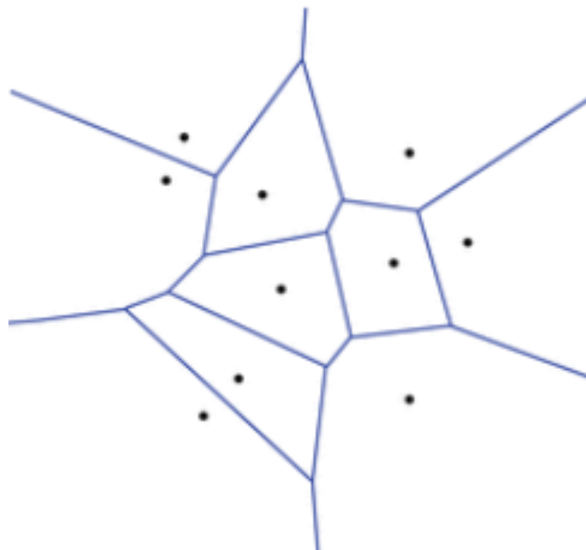


Figure 3.2: An example voronoi diagram. For NNeW, the points in this illustration are the source samples. [57]

competitiveness compared to other state-of-the-art importance weight estimators. However, its performance suffers from cases in which a lot of source samples are assigned a zero weight.

To combat this, the Laplace smoothed variant NNeW+1 was introduced, which initializes the count for each cell V_i with 1. Thus the weights are estimated to be proportional to

$$\hat{w}(x_i) = |V_i \cap \{\xi_j\}_{j=1}^{N_\tau}| + 1. \tag{17}$$

This variant proved to have slightly better performance while eliminating the problem of zero weights. Therefore, the implementation used in this paper [58] uses NNeW+1 instead of NNeW.

3.2.2 Logistic Discrimination

Like NNeW, the Logistic Discrimination [55] approach is an importance weighting method. In this method, the desired weight for each source sample is the ratio between the probability that the target domain assigns to the sample ($p_t(x) = p(x|\theta)$) and the probability that the source domain assigns to the sample ($p_s(x) = p(x|\lambda)$). However, Bickel et al. [55] aim to do this without estimating the source and target distributions explicitly. In their research, they present following equation to determine the optimal weights $\frac{p_t(x)}{p_s(x)}$ with a logistic model, without knowing the source or target density:

$$\frac{p(s=1)}{p(s=-1)} \exp(-v^T x). \tag{18}$$

$p(s = 1)$ and $p(s = -1)$ are the marginal probabilities of observing a sample from the target and the source domain which can be ignored since the ratio is constant and therefore all samples will be scaled the same. v is the parameter vector of a logistic regression classifier trained to distinguish both domains from each other and x is a source/target sample. So Bickel et al. propose to train a logistic model in distinguishing source from target samples and then use the parameter vector of this model to determine the weights of each source sample as an approximation of $\frac{p_t(x)}{p_s(x)}$.

3.2.3 Transfer Component Analysis

Unlike the two previous approaches, Transfer Component Analysis (TCA), proposed by Pan et al. in [8], does not rely on importance weighting. Instead, TCA tries to find some kind of transformation ϕ on the source and target data so that $P(\phi(X_S)) \approx P(\phi(X_T))$. However, since the unsupervised learning setting does not provide any labels in the target domain, this transformation ϕ cannot be directly learned. Therefore, Pan et al. propose to learn ϕ so that it satisfies two conditions:

1. The distance between the source and target distributions is small.
2. Important properties of X_S and X_T are preserved in the transformed datasets $\phi(X_S)$ and $\phi(X_T)$.

To achieve this goal, TCA aims to find so called *transfer components* that underly both domains. For two related domains, there may be a number of common components so TCA tries to find those components that make the difference between the data distributions small, while also maintaining the discriminative features of the untransformed data. After finding such transfer components, the original data is projected onto the subspace spanned by these components. Following this projection, any method can be used for classification.

Chapter 4

Experiments

For this thesis, I conducted multiple experiments with the afore mentioned models. This chapter will give you an overview on how these experiments were conducted, how the models were variated and fine-tuned, how the results were compared and lastly, which results were achieved and how each model compares to each other.

4.1 Setup

Before I outline which models were tested and how they were tested, I would like to explain some of the setup for the baseline domain adaptation methods (i.e. cross-validation) and the measures by which the models will be compared to each other and to the work of Buschjäger et al. [2], the basis of this thesis.

4.1.1 Cross-Validation

Cross-validation can be seen as a method to divide a given dataset into a calibration (used for training) and a validation set (used for validating/testing) [59]. This is important when training machine learning models to prevent problems such as overfitting (i.e. a model that fits very/too precisely to the training dataset, so that it cannot produce reliable predictions on test data [60]). There exist several approaches outlining how to divide the dataset into the calibration and validation set and a number of papers have been published on this topic. In the following I will be presenting some popular methods, including the one used in my baseline experiments.

A very simple cross-validation method is the *leave-one-out* method. Here, a dataset of N samples is divided into a $N-1$-sized calibration set and a validation set containing just one sample. The model is then trained on the calibration set and validated with the validation set, which means that a performance score is computed on the validation set. This process is repeated N times so that every sample is used as the validation sample exactly once. Then an average of the N performances is used [59]. This can also

be extended to the *leave-some-out* method, where instead of using only one sample in the validation set, the validation set is made up out of a previously specified number of samples. The calibration set then consists out of $N - n$ samples, with n being the amount of samples left out at every iteration.

A different method would be the *Monte Carlo* (or *random subsampling*) cross-validation (MCCV)[61], where the dataset is randomly split into a calibration and validation set. The process is repeated multiple times and the results are averaged. Each CV-iteration could produce different split ratios, as MCCV does not randomly assign a fixed number of samples to either the calibration or validation set, but rather iterates other each sample and randomly assigns them. This results in a random amount of randomly selected samples in each CV-set. In this method, there is no guarantee for each sample being in the validation set at least once and it is also possible for a sample to be in the validation set more than once.

The cross-validation method that I will be using, first brought forward by Geisser [62, 61], is the so-called *V-fold* cross-validation (with $V \in \{1, ..., n\}$ and n denoting the amount of samples in the dataset). For this approach, the data is partitioned into V subsamples of equal size and each of these subsamples is used as the validation set exactly once while the other subsamples form the calibration set for that iteration. Therefore the model is trained a total of V times and just like the other methods the result is the average of the results from each of the V models. Each of the V subsamples is made up out of $\frac{n}{V}$ randomly selected samples. Unlike MCCV, this cross-validation method ensures that each sample is used for validation exactly once while being used for calibration during the remaining CV-iterations.

All of the cross-validation methods outlined above perform their calibration-validation splits other multiple iterations. In each of these iterations the accuracy of the model trained on the respective calibration set is calculated over the predictions made with the validation set. In our setting, the accuracy is calculated slightly differently.

4.1.2 Accuracy

An easy way to determine how good a specific machine learning model actually is, and a way to compare different models, is calculating its predictive accuracy. In a basic machine learning setting this can be easily done by feeding an unseen test set of labeled data into the model and calculating the respective outputs. From these calculated outputs one can in turn derive the predicted class of each data point from the test set and compare this to the actual class labels. The accuracy then denotes the fraction of correct classifications.

In our case, much like unsupervised domain adaptation settings in general, the aim is to train a model to mainly achieve good predictive accuracy on the target data. So while one could still evaluate the models accuracy on the labeled source data, which might also

be desired in some case, the more important measure would be the accuracy on the target data, which is not as easy to calculated in unsupervised DA settings as the available target data is unlabeled. Therefore one must find some sort of workaround in order to determine the models performance on the target data. Accordingly, I will be using such a workaround in my experiments, specifically I used the method outlined in [63].

The problem with calculating the accuracy on the target data are the missing class labels. Without these, one cannot verify the predictions made by the model. To combat this, Bruzzone et al. [63] proposes to interpret the predicted class labels as the actual class labels for the respective data points from the target distribution. Then, with these "pseudo-labels", the model can be trained with the target test set (including the predicted labels) acting as the source domain. The labeled training dataset, which was used to first train the model, is in turn used for the domain adaptation (omitting its labels). In short, the source and target domains are switched, and the model is retrained. This retrained model is then used to make predictions on the labeled test set and the accuracy is computed over these predictions [63].

In their work, Bruzzone et al. found that this "circular validation strategy" was able to identify acceptable solutions for the target domain and reject those solutions inconsistent with the target domain problem [63]. Nevertheless, some of the consistent solutions might also be rejected, since a perfect symmetry between the two domain adaptation problems (from source to target and from target to source) cannot be assumed. However, several tests conducted in [63] confirmed that this validation strategy provides good estimates for the accuracy of domain adaptation models.

4.1.3 Performance Metric

In Buschäger et al. [2], instead of determining the performance of their model by calculating the accuracy on the real data as explained in the section above, the performance on the real data is measured by the significance of detection S_{LiMa} according to the work conducted in [64]. To calculate this measure, the direction of the gamma rays is estimated and the angle between the incoming ray and the direction of the Crab Nebula is computed. The number of gamma rays is then regarded as a distribution depending on the angle. The distribution with respect to the direction of the Crab Nebula then builds the on-distribution N_{on}, while the distributions for five positions without any known gamma sources make up the off-distributions N_{off} [2]. If there was no gamma source in the direction of the Crab Nebula, the ratio of the expected counts is a known value α. Therefore, for a positive identification of gamma rays $N_{on} > \alpha N_{off}$ is required. The excess counts of gamma rays is in turn defined as

$$N_S = N_{on} - \alpha N_{off}. \tag{19}$$

The significance of detection S_{LiMa} is then reported by the standard deviation σ of N_S [2, 64, 65].

This is the performance metric I will be using, where higher values are better. It should be noted that random predictions achieve values around 5 and good predictions produce values around 20.

4.2 Deep Unsupervised Domain Adaptation

For the deep unsupervised domain adaptation approach, the neural network used in [2] was extended based on the research conducted by Ganin et al. in [1]. This means that the basic structure of the CNN (i.e. feature extractor and label predictor) was kept identical to the structure in [2]. In order to incorporate the domain classifier from [1], a second fully-connected layer was added via a gradient reversal layer (GRL).

As a first experiment, the domain classifier was given the same structure as the label predictor (one linear layer of size 512: $x \rightarrow 512 \rightarrow 2$). Henceforth this model will be denoted as `DA-512`. To find the best possible solution, the meta-parameters of `DA-512` were tweaked. However, for each tested model only one parameter was changed so as to more accurately see the effect of this particular meta-parameter. If multiple parameters were changed at a time, it would be difficult to determine wether the change in performance was cause by one certain parameter change or by the combination of multiple parameter changes.

In a first attempt to optimize the networks parameters, the size of the domain classifier was reduced to ($x \rightarrow 128 \rightarrow 2$) (`DA-128`). Later on, the size of the domain classifier was extended to ($x \rightarrow 512 \rightarrow 512 \rightarrow 2$) (`DA-512-512`). For another model, the parameter γ (from eq. 15) was adjusted to change the influence of the domain classifier in the early stages of training. In [1], γ was set to 10 and not further adjusted. This value was adopted for the three afore mentioned models and only changed for the following models. A smaller value for γ results in the domain classifier having less influence at the beginning of training and it takes more epochs for the domain classifier to have the same influence as the label predictor. A greater value for γ, intuitively, has the opposite effect. From now on, the models with adjusted γ values will be referenced as `DA-`γ_x, with x representing the value assigned to γ.

All of the afore mentioned models were trained with the simulated training data produced by CORSIKA [52], referenced in section 3.1, as well as real-world data colltected by FACT (section 2.4). The simulated data is the same data as used by Buschjäger et al. in [2]. The simulation data is made up of 200,000 training samples and 100,000 test samples. Both the training set and the test set present balanced amounts of gamma and hadron events [2]. Furthermore, two variants of these simulated sets were used in [2]: firstly with quality cuts and secondly without quality cuts. In the set with quality cuts, unreal-

istic simulated events were eliminated. My adapted models were trained on both sets of data, yet the comparison with the Buschjäger approach will be conducted over the models trained on the dataset without quality cuts, as this dataset produced the best results in [2].

The real-world data was once obtained when the telescope was directed at the Crab Nebula, a known source of gamma rays [2]. This dataset consists of 3,972,043 samples/events. However, due to our training setup, only 300,000 samples were used at training time. These 300,000 events were randomly selected and randomly split into a training set and a test/validation set containing 200,000 events and 100,000 events, respectively. All models were trained on 128-sized batches, each batch containing 64 simulated events and 64 real events.

After each epoch, the simulated test set and the real test set were used for validation and the accuracy of the label predictor (only measured on the simulated validation set due to the lack of labels for the real data) and domain classifier was measured. These accuracies were then compared to the previous epochs to save the best model. During training, three different epochs were saved: the one with the best label predictor accuracy, the best domain classifier accuracy and the best overall accuracy. While the best label predictor accuracy is the highest accuracy encountered, the best domain classifier accuracy is actually the worst accuracy encountered. Since the aim is to find domain-invariant features, the domain classifier should not be able to differentiate between the two domains. Therefore the optimal domain classifier accuracy would be 0.5, because an accuracy of 0 would mean that every sample is classified to the correct domain and an accuracy of 1 would mean that every sample is classified to the wrong domain, so every source sample is classified as a target domain sample and vice versa, meaning the classifier is still able to differentiate between the two domains. Thus, I will be using "best domain classifier accuracy" as a synonym for "the accuracy closest to 0.5". Since it is possible for the epoch with the best label predictor accuracy to achieve a bad domain classifier accuracy (and vice versa), resulting in bad predictions in the target domain, I also calculated the best overall accuracy, which was simply done by adding the label predictor loss to the variance to 0.5 of the domain classifier accuracy. Then the smallest value was saved as the best overall loss.

The loss functions used for each model was cross-entropy for both the label predictor and the domain classifier. Just like in [2], the ASMGrad optimizer was used. While Buschjäger et al. used an initial learning rate of 0.001 which was reduced by a factor of 0.1 every 25 epochs, in my experiments the initial learning rate was set to 0.005 and it was reduced every epoch, but in a way that the same value as in [2] was reached every 25 epochs.

In the first few experiments, the models were trained for 100 epochs, however it could be seen, that the models were overfitted. Therefore, the later models were trained over 50 epochs, as this proved to be sufficient in finding the model with the best validation loss.

For all models, the three saved epochs were compared based on the significance of detection S_{LiMa}.

4.3 Baselines

Since the aim of this paper is not only to compare a deep domain adaptation approach to a deep model without domain adaptation, but also to analyze the performance of this deep DA approach to other DA approaches, I tested multiple baseline methods as explained in section 3.2. Each of these methods was trained on a dataset containing hand-crafted high-level features (for simulated data as well as real data). However, the amount of data used for training differed for each method due to computational complexity, so that the resource footprint (runtime, memory allocation) was comparable among all baseline methods. For all methods, a 5-fold cross-validation (section 4.1.1) was applied. For each cross-validation iteration, the model was trained on a training set of the simulation and real data, the performance was measured (section 4.1.3) and the accuracy was calculated (section 4.1.2). The overall accuracy of each model was the average accuracy achieved across all cross-validation iterations.

Two of the three selected baseline methods rely on importance weighing (each source sample is assigned a weight with the help of the target samples and the weighted samples are fed to some sort of classifier) (sections 3.2.1 and 3.2.2). For these two methods, two different classifiers were tested: a logistic regression classifier (LR) and a random forest classifier(RF). The random forest classifier used consists of 100 trees without a maximum depth.

4.4 Implementation

For the baseline methods, I utilized the implementations given in LibTLDA [58] which provided basic implementations of a number of domain adaptation methods. One of these methods, namely Logistic Discrimination, was not implemented correctly and therefore returned very bad results. To combat this, some changes had to be made. I implemented a logistic regression classifier to differentiate between source and target domain data (simulation and real data) and used its parameter vector to determine the weights of the source samples as described in section 3.2.2. This implementation provided good results, however some hadron samples were assigned very large weights. This made the predictions too inconsistent. These weights are a result of the different distributions in the source and target domain. In the source domain, there is an equal distribution of gammas and

hadrons. However, in the target domain, the are significantly less gamma events than hadron events. Thus the logistic regression classifier often misclassifies hadron samples from the source domain, not for being close to the source domain but purely for the fact it is a hadron, and as a result these hadron sample is assigned a large weight.

While the random forest classifier (which is trained on the weighted samples) seemed to be robust to these outliers and produce constant results, the logistic regression classifier (not the one used to determine the weights but the one used for gamma classification) was susceptible to these high weights, occasionally returning an accuracy of around 50% while still producing acceptable values in the Li&Ma Test (section 4.1.3).

To combat this problem, I randomly flipped a certain percentage of the domain labels of the target domain data before training the classifier and calculating the weights. While this did lead to more stable accuracies, the average accuracy stayed the same and the σ values decreased in some cross-validation iterations. I tested this method with different percentages, with 17% providing the best trade-off between stable accuracies and good σ values.

Since the random forest classifier did not have any problems with too large weights, this approach was implemented without flipping any domain labels.

All of the other baseline methods were kept the same as in [58].

The CNNs used in the deep domain adaptation experiments were implemented using PyTorch[1].

[1] https://www.pytorch.org

4.5 Results

The results will be discussed in three separate parts. First, the results of the baseline domain adaptation methods will be presented and compared amongst each other. Then the same will be done for the deep domain adaptation models. Finally, my findings will be compared to the model presented by Buschjäger et al. [2].

4.5.1 Baselines

In this section I will not only present the results of the tested baseline methods, but also those of the classifiers used in the importance weighting approaches (without domain adaptation) to see how the results changed with the addition of the domain adaptation elements. The average accuracies of all baseline methods, together with the amount of data used for training, can be seen in Table 4.1. When looking at the table, one immediately notices the bad performance of TCA. In my experiments, TCA could only be trained with 2,000 samples from the source domain and 2,000 samples from the target domain which resulted in an average accuracy of only 0.5025. The reason that only 2,000 samples from each domain were used is that TCA makes use of multiple kernel matrices, which needed too much memory when presented with more than 2,000 samples.

The NNeW method of calculating sample weights was not able to improve the accuracy of the LR classifier, however it was only trained with $\frac{1}{5}$ as many samples as the LR classifier and the accuracy achieved was only marginally smaller. It would be interesting to see how the accuracy of the NNeW approach changes, when more data is used for training. Since the LibTLDA [58] implementation calculates the NNeW-weights using a $m \times n$-matrix (m being the amount of source samples and n the amount of target samples), no more than 100,000 samples could be used due to limited memory availability. When combined with

Method	Datasize	Avg. Accuracy	
		LR	RF
LD	300,000	**0.6146**	0.6803
NNeW	100,000	0.6136	6713
TCA	2,000	0.5025	
LR	500,000	0.6143	
RF	300,000		**0.6805**

Table 4.1: Accuracy on the real data, calculated according to section 4.1.2. The tested methods are: Logistic Discrimination (LD), Nearest-Neighbor Importance Weighting (NNeW), Transfer Component Analysis (TCA), Logistic Regression (LR) and Random Forest (RF). For the importance weighting approaches (LD and NNeW) we differentiate between LR and RF (the classifiers used for gamma classification). The best results for each classifier (LR and RF) are highlighted.

the RF classifier, NNeW again achieved slightly worse accuracies than the random forest classifier without domain adaptation.

The best performing baseline method I tested was the logistic discrimination approach. It was able to slightly increase the predictive performance of the LR classifier and while the improvement is only minimal, it was achieved with less data used for training. In combination with the random forest classifier, LD was not able to improve the predictive accuracy. The RF classifier without domain adaptation actually achieved the best average accuracy of all baselines tested, with LD only slightly behind.

In general it could be seen that the logistic regression classifier was not able to get close to the accuracies of the random forest classifier (both with and without domain adaptation).

In order to compare the baseline methods according to the significance of detection S_{LiMa}, I will be choosing the best σ value achieved within all cross-validation iterations for all methods. The respective results are shown in Table 4.2.

The TCA domain adaptation method is the worst performing model with regards to the accuracy and S_{LiMa}. After seeing the achieved accuracy, a significance of detection of 11.71σ was no surprise since this is about the same a model with random predictions would achieve (a model with random predictions also has an accuracy of around 0.5). With regards to the logistic regression classifier, the best performing model was the one using NNeW, achieving a higher S_{LiMa} than the LR classifier trained with more data but without domain adaptation. Similar to the accuracies, the significance of detection is higher for every method when the classification was made by a random forest classifier rather than a logistic regression classifier.

<table>
<tr><th rowspan="2">Method</th><th rowspan="2">Datasize</th><th colspan="2">S_{LiMa}</th></tr>
<tr><th>LR</th><th>RF</th></tr>
<tr><td>LD</td><td>300,000</td><td>18.60σ</td><td>$\mathbf{20.06}\sigma$</td></tr>
<tr><td>NNeW</td><td>100,000</td><td>$\mathbf{19.75}\sigma$</td><td>19.97σ</td></tr>
<tr><td>TCA</td><td>2,000</td><td colspan="2">11.71σ</td></tr>
<tr><td>LR</td><td>500,000</td><td>18.69σ</td><td></td></tr>
<tr><td>RF</td><td>300,000</td><td></td><td>19.73σ</td></tr>
</table>

Table 4.2: Significance of detection S_{LiMa}, calculated according to section 4.1.3. The tested methods are: Logistic Discrimination (LD), Nearest-Neighbor Importance Weighting (NNeW), Transfer Component Analysis (TCA), Logistic Regression (LR) and Random Forest (RF). For the importance weighting approaches (LD and NNeW) we differentiate between LR and RF (the classifiers used for gamma classification). The best results for each classifier (LR and RF) are highlighted.

Even though both the LD and NNeW approach achieved lower average accuracies than the RF classifier without domain adaptation, both produced higher σ values, with LD achieving the highest significance of detection of all baseline methods.

4.5.2 Deep Unsupervised Domain Adaptation

Since the accuracy calculation for the real data involves retraining a new model from scratch, the CNNs tested in this thesis were compared to each other using the significance of detection. Then the accuracy was calculated for the best model and later compared to the best baseline model and the model put forward by Buschjäger et al..

As explained in section 4.2, the models from three epochs were saved for each tested CNN and the significance of detection was calculated for each of these epochs. The results of the different deep domain adaptation models tested are presented in Table 4.3

In my experiments it became clear that a good accuracy of the label predictor was more important for good predictions on the target domain than a good domain classifier accuracy. This can be seen in Table 4.3, since all models but one achieved their best significance of detection in the epoch with the best label predictor loss. The best performing model, however, actually produced the best results in the epoch with the best domain classifier loss, which in all other models was the worst of the three tested epochs. The reason for this is that the epoch with the best label predictor accuracy (LPA) performed poorly in terms of domain classifier accuracy (DCA). While the desired DCA is 0.5, this epoch only achieved 0.6833, meaning that this epoch found features that provided useful information in the source domain, but these features were not as relevant in the target domain. In the epoch with the best DCA, the DCA was 0.5017 and the LPA was not much worse than in the epoch with the best label loss (0.8724 and 0.9054, respectively). This combination lead to the best significance of detection across all models I tested.

Almost all models with adjusted parameters were able to improve the "basic" approach of `DA-512`. The only model with a worse performance was `DA-512-512`. The extended domain classifier, which was able to achieve the best domain classifier accuracy of all

Model	S_{LiMa}		
	epoch: best label loss	epoch: best domain loss	epoch: best overall loss
`DA-512`	26.29σ	24.35σ	26.28σ
`DA-128`	25.19σ	**28.76σ**	27.20σ
`DA-512-512`	26.05σ	25.30σ	25.30σ
`DA-`γ_{20}	26.93σ	22.85σ	22.85σ
`DA-`γ_5	27.12σ	23.14σ	26.45σ

Table 4.3: Significance of detection S_{LiMa}, calculated according to section 4.1.3. The tested CNNs are outlined in section 4.2. The best significance of detection is highlighted.

models tested (DCA: 0.4992), actually decreased the performance of the label predictor. This is because the calculated domain loss is carried further into the network during the optimization of the networks weights and biases (due to the extended size of the domain classifier), resulting in a worse performing label predictor. A similar effect can be seen with `DA-`γ_{20}. Though this model actually improved `DA-512`, this was probably an effect caused by the random initialization of weights at the start of training, since the model with the best LPA was found within the first 10 Epochs.

While the best significance of detection gives us a clear ranking of the models, the best way to compare the effects of each adjustment is to look at the epoch with the best overall loss. In this epoch the best trade-off was made between the LPA and DCA. As can be seen in Table 4.3, the models that decrease the influence of the domain classifier into finding the most relevant features (`DA-128` and `DA-`γ_5) achieve a greater S_{LiMa} in the epoch with the best overall loss than the reference model `DA-512`. The models that increase the influence of the domain classifier (`DA-512-512` and `DA-`$\gamma_2$0), on the other hand, perform worse than `DA-512`. This confirms my first observation of the label predictor accuracy being more important for good predictions in the target domain than the domain classifier accuracy.

As mentioned in section 4.1.3, a good model should achieve a significance of detection of around 20σ. With this in mind, all models can be seen as good predictors for the target domain. But how do they compare to a neural network trained without adaptation as proposed by Buschjäger et al.?

4.5.3 Comparison

Table 4.4 presents the significance of detection as well as the accuracy for the best models encountered in my experiments and the model developed in [2]. To ensure that my results are comparable to the ones achieved by Buschjäger et al., I tried to reproduce their results and achieved a comparable significance of detection (in my test their model achieved 25.05σ while their tests produced 24.64σ). To calculate the accuracy of their model I used the epoch with the best validation loss to make predictions on the real data and trained the

Model	S_{LiMa}	Accuracy
LD-RF	20.06σ	0.6803
`DA-128`	$\mathbf{28.76}\sigma$	0.8885
Buschjäger et al. [2]	24.64σ	**0.9004**

Table 4.4: Significance of detection S_{LiMa} and accuracy of the best performing models and the non-domain adaptation approach. The best significance of detection and accuracy is highlighted. LD-RF denotes the logistic discrimination approach to importance weighting in combination with a random forest classifier.

model with the real data and these predicted labels. Then the accuracy was calculated over the simulated test set with the epoch with the best validation loss.

Let us first have a look at the significance of detection. As can be seen, the two neural networks easily outperform the shallow domain adaptation method. What's more, `DA-128` presents a noticeable improvement over the model developed by Buschjäger et al.. Looking back at Table 4.3, one can actually see that all tested deep domain adaptation models outperformed the model trained purely on simulation data and therefore confirm the effectiveness of the method proposed by Ganin et al. [1].

When comparing the accuracies of each model, similar conclusions can be made. Once again, the shallow domain adaptation models are the worst performing models. However, the deep domain adaptation model achieves a slightly lower accuracy than the non-DA model. The reason for this could lie in the way the accuracy is calculated. Since the accuracy is calculated in a circular workaround (section 4.1.2), the resulting accuracy is only an estimate and might reject some good solutions. Nonetheless, the calculated accuracy for the model designed by Buschjäger et al. is the highest. To further examine these calculated accuracies I also tested the accuracy on only hadron events and only gamma events.

By looking at Table 4.5, one can easily see the reason for the difference in overall accuracy. While the Buschjäger et al.-model achieves similar accuracies for both gamma and hadron samples, `DA-128` achieves much higher accuracies on the set containing only hadron events than on the one containing only gamma events. The reason for these results most probably lies in the different distributions of gamma and hadron events across the two domains. Since the simulation data provides an equal amount of gamma and hadron events, the model trained purely on simulation data also achieves similar accuracies for both classes of events. In the real data, the hadron events significantly overweigh the gamma events, resulting in `DA-128` being more confident in its hadron predictions than in its gamma predictions.

Since the accuracy measure used in this thesis comes with small problems (occasionally rejecting good solutions), the better measure to compare the gamma-hadron separation models against each other is the significance of detection S_{LiMa}, in which the deep domain adaptation model is the clear winner.

Model	Accuracy on h.e.	Accuracy on g.e.	Ovr. Accuracy
`DA-128`	**0.9577**	0.8193	0.8885
Buschjäger et al. [2]	0.9058	**0.8949**	**0.9004**

Table 4.5: The accuracy of the two deep machine learning models. "h.e." stands for "hadron events" and "g.e." stands for "gamma events".

Chapter 5

Summary and Outlook

In this thesis I tested several domain adaptation methods with the aim of improving predictive capabilities when confronted with unseen and unlabeled telescope observations of possible gamma events. Instead of being trained solely on simulated data, the tested models should be confronted with real data to compensate for the different marginal probability distributions of simulated and real data. To examine the effectiveness of domain adaptation models in the setting of gamma-hadron separation, different approaches were taken. First, several "shallow" methods were tested, such as Nearest-Neighbor Importance Weighting (NNeW), Logistic Discrimination (LD) and Transfer Component Analysis (TCA). However none of these methods were able to significantly improve the performance of their underlying classifiers. Nonetheless, the results obtained, except for the TCA results, were acceptable considering the computational complexity of these methods. The results of the TCA method were rather disappointing due to the large amount of memory needed.

To further examine the effectiveness of domain adaptation, an established CNN model for gamma-hadron separation [2] was extended by a second fully-connected layer (domain classifier), based on the work of Ganin et al. [1], in order to incorporate real unlabeled data into the training process. This form of deep domain adaptation proved to be very effective, with every tested model outperforming the model proposed in [2] by Buschjäger et al. (in terms of S_{LiMa}). It was shown that while the domain classifier in each of the tested models lead to predictions being made based on more domain-invariant features, some domain classifiers also found more discriminative features. Through a small amount of fine-tuning, it was discovered that the best trade-off between domain-invariant and discriminative features was found when the influence of the domain classifier was smaller. This way the CNN could first focus on finding discriminative features before progressively focusing on those that were also domain-invariant.

In short, the work conducted in this thesis confirmed that neural networks trained for gamma-hadron separation can in fact be improved by means of domain adaptation. Additionally, this thesis showed that, for this particular task, deep domain adaptation models are far superior than any shallow method. In the future, it would be interesting to see how the CNNs tested in this thesis can be improved by further adjustments of the networks meta-parameters. Since the shrinking of the layer size (in the domain classifier) and the decreasing of the γ value (Eq. 15) both, individually, showed improvements to the untouched model (`DA-512`), one could start by combining these adjustments to possibly further improve predictive capabilities.

List of Figures

2.1 (a) illustrates the basic sigmoid for a neuron with a single input. In this case the weight of the input was fixed at 1 and the bias was set to 0. Changing the bias would result in a shifting of the sigmoid along the x axis, while a change in the weight would affect the steepness of the function [12]. (b) depicts the popular ReLU activation function. The functions were plotted using FooPlot[1]. 6
2.2 A basic representation of a convolutional neural network architecture [22]. . 9
2.3 The convolution process. (a) depicts where the filter is applied to the image with stride 2 and where the results are saved in the feature map. (b) shows how the filter is applied to a certain patch of the image by calculating the dot product [22]. 10
2.4 The max pooling process. Each colored section depicts a patch covered by the filter. For each patch, the filter returns its maximum value [22]. 11
2.5 An illustration of the emergence of Cherenkov light that is in turn detected by the telescope (left) and a camera sample from said telescope (right). [2]. 15

3.1 The proposed CNN architecture for deep unsupervised domain adaptation by Ganin et al. [1]. 18
3.2 An example voronoi diagram. For NNeW, the points in this illustration are the source samples. [57] 20

List of Tables

4.1 Accuracy on the real data, calculated according to section 4.1.2. The tested methods are: Logistic Discrimination (LD), Nearest-Neighbor Importance Weighting (NNeW), Transfer Component Analysis (TCA), Logistic Regression (LR) and Random Forest (RF). For the importance weighting approaches (LD and NNeW) we differentiate between LR and RF (the classifiers used for gamma classification). The best results for each classifier (LR and RF) are highlighted. 30

4.2 Significance of detection S_{LiMa}, calculated according to section 4.1.3. The tested methods are: Logistic Discrimination (LD), Nearest-Neighbor Importance Weighting (NNeW), Transfer Component Analysis (TCA), Logistic Regression (LR) and Random Forest (RF). For the importance weighting approaches (LD and NNeW) we differentiate between LR and RF (the classifiers used for gamma classification). The best results for each classifier (LR and RF) are highlighted. 31

4.3 Significance of detection S_{LiMa}, calculated according to section 4.1.3. The tested CNNs are outlined in section 4.2. The best significance of detection is highlighted. 32

4.4 Significance of detection S_{LiMa} and accuracy of the best performing models and the non-domain adaptation approach. The best significance of detection and accuracy is highlighted. LD-RF denotes the logistic discrimination approach to importance weighting in combination with a random forest classifier. 33

4.5 The accuracy of the two deep machine learning models. "h.e." stands for "hadron events" and "g.e." stands for "gamma events". 34

References

[1] Yaroslav Ganin and Victor Lempitsky. "Unsupervised Domain Adaptation by Backpropagation". In: Proceedings of Machine Learning Research 37 (July 2015). Ed. by Francis Bach and David Blei, pp. 1180–1189. URL: http://proceedings.mlr.press/v37/ganin15.html.

[2] Sebastian Buschjäger et al. "On-Site Gamma-Hadron Separation with Deep Learning on FPGAs". In: (2020).

[3] S. J. Pan and Q. Yang. "A Survey on Transfer Learning". In: *IEEE Transactions on Knowledge and Data Engineering* 22.10 (2010), pp. 1345–1359. DOI: 10.1109/TKDE.2009.191.

[4] Garrett Wilson and Diane J. Cook. "A Survey of Unsupervised Deep Domain Adaptation". In: *ACM Trans. Intell. Syst. Technol.* 11.5 (July 2020). ISSN: 2157-6904. DOI: 10.1145/3400066. URL: https://doi.org/10.1145/3400066.

[5] Karsten M. Borgwardt et al. "Integrating structured biological data by Kernel Maximum Mean Discrepancy". In: *Bioinformatics* 22.14 (July 2006), e49–e57. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btl242. eprint: https://academic.oup.com/bioinformatics/article-pdf/22/14/e49/616383/btl242.pdf. URL: https://doi.org/10.1093/bioinformatics/btl242.

[6] Jiayuan Huang et al. "Correcting sample selection bias by unlabeled data". In: *Advances in neural information processing systems* 19 (2006), pp. 601–608.

[7] Boqing Gong, Kristen Grauman, and Fei Sha. "Connecting the dots with landmarks: Discriminatively learning domain-invariant features for unsupervised domain adaptation". In: *International Conference on Machine Learning*. PMLR. 2013, pp. 222–230.

[8] Sinno Jialin Pan et al. "Domain adaptation via transfer component analysis". In: *IEEE Transactions on Neural Networks* 22.2 (2010), pp. 199–210.

[9] Andries P Engelbrecht. *Computational intelligence: an introduction*. John Wiley & Sons, 2007.

[10] Hasan Tercan et al. “Transfer-learning: Bridging the gap between real and simulation data for machine learning in injection molding”. In: *Procedia Cirp* 72 (2018), pp. 185–190.

[11] Paul J Werbos. “Backpropagation through time: what it does and how to do it”. In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560.

[12] Barry J Wythoff. “Backpropagation neural networks: a tutorial”. In: *Chemometrics and Intelligent Laboratory Systems* 18.2 (1993), pp. 115–155.

[13] P Sibi, S Allwyn Jones, and P Siddarth. “Analysis of different activation functions using back propagation neural networks”. In: *Journal of theoretical and applied information technology* 47.3 (2013), pp. 1264–1268.

[14] Prajit Ramachandran, Barret Zoph, and Quoc V Le. “Searching for activation functions”. In: *arXiv preprint arXiv:1710.05941* (2017).

[15] Elisa Oostwal, Michiel Straat, and Michael Biehl. “Hidden unit specialization in layered neural networks: ReLU vs. sigmoidal activation”. In: *Physica A: Statistical Mechanics and its Applications* 564 (2021), p. 125517.

[16] Katarzyna Janocha and Wojciech Marian Czarnecki. “On loss functions for deep neural networks in classification”. In: *arXiv preprint arXiv:1702.05659* (2017).

[17] Pushparaja Murugan. “Feed forward and backward run in deep convolution neural network”. In: *arXiv preprint arXiv:1711.03278* (2017).

[18] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[19] Ahmed Ali Mohammed Al-Saffar, Hai Tao, and Mohammed Ahmed Talab. “Review of deep convolution neural network in image classification”. In: *2017 International Conference on Radar, Antenna, Microwave, Electronics, and Telecommunications (ICRAMET)*. IEEE. 2017, pp. 26–31.

[20] Jie Lu et al. “Transfer learning using computational intelligence: A survey”. In: *Knowledge-Based Systems* 80 (2015), pp. 14–23.

[21] Yann LeCun, Yoshua Bengio, et al. “Convolutional networks for images, speech, and time series”. In: *The handbook of brain theory and neural networks* 3361.10 (1995), p. 1995.

[22] Boukaye Boubacar Traore, Bernard Kamsu-Foguem, and Fana Tangara. “Deep convolution neural network for image recognition”. In: *Ecological Informatics* 48 (2018), pp. 257–268.

[23] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444.

[24] Kamran Kowsari et al. "Text classification algorithms: A survey". In: *Information* 10.4 (2019), p. 150.

[25] Xiang Zhang, Junbo Zhao, and Yann LeCun. "Character-level convolutional networks for text classification". In: *arXiv preprint arXiv:1509.01626* (2015).

[26] Charu C Aggarwal and ChengXiang Zhai. "A survey of text classification algorithms". In: *Mining text data*. Springer, 2012, pp. 163–222.

[27] Arushi Gupta and Rishabh Kaushal. "Improving spam detection in online social networks". In: *2015 International conference on cognitive computing and information processing (CCIP)*. IEEE. 2015, pp. 1–6.

[28] Nitin Jindal and Bing Liu. "Review spam detection". In: *Proceedings of the 16th international conference on World Wide Web*. 2007, pp. 1189–1190.

[29] Dengsheng Lu and Qihao Weng. "A survey of image classification methods and techniques for improving classification performance". In: *International journal of Remote sensing* 28.5 (2007), pp. 823–870.

[30] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. "Multi-column deep neural networks for image classification". In: *2012 IEEE conference on computer vision and pattern recognition*. IEEE. 2012, pp. 3642–3649.

[31] Waseem Rawat and Zenghui Wang. "Deep convolutional neural networks for image classification: A comprehensive review". In: *Neural computation* 29.9 (2017), pp. 2352–2449.

[32] Lei Xu, Adam Krzyzak, and Ching Y Suen. "Methods of combining multiple classifiers and their applications to handwriting recognition". In: *IEEE transactions on systems, man, and cybernetics* 22.3 (1992), pp. 418–435.

[33] Claus Bahlmann, Bernard Haasdonk, and Hans Burkhardt. "Online handwriting recognition with support vector machines-a kernel approach". In: *Proceedings Eighth International Workshop on Frontiers in Handwriting Recognition*. IEEE. 2002, pp. 49–54.

[34] Jui-Ting Huang et al. "Cross-language knowledge transfer using multilingual deep neural network with shared hidden layers". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE. 2013, pp. 7304–7308.

[35] Nidhi Desai, Kinnal Dhameliya, and Vijayendra Desai. "Feature extraction and classification techniques for speech recognition: A review". In: *International Journal of Emerging Technology and Advanced Engineering* 3.12 (2013), pp. 367–371.

[36] Keunwoo Choi et al. "Transfer learning for music classification and regression tasks". In: *arXiv preprint arXiv:1703.09179* (2017).

[37] Changsheng Xu, Namunu Chinthaka Maddage, and Xi Shao. "Automatic music classification and summarization". In: *IEEE transactions on speech and audio processing* 13.3 (2005), pp. 441–450.

[38] Ritanjali Majhi et al. "Classification of consumer behavior using functional link artificial neural network". In: *2010 International Conference on Advances in Computer Engineering*. IEEE. 2010, pp. 323–325.

[39] Tobias Lang and Matthias Rettenmeier. "Understanding consumer behavior with recurrent neural networks". In: *Workshop on Machine Learning Methods for Recommender Systems*. 2017.

[40] Adam Page, Colin Shea, and Tinoosh Mohsenin. "Wearable seizure detection using convolutional neural networks with transfer learning". In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2016, pp. 1086–1089.

[41] Rich Caruana. "Multitask learning". In: *Machine learning* 28.1 (1997), pp. 41–75.

[42] Ioannis D Apostolopoulos and Tzani A Mpesiana. "Covid-19: automatic detection from x-ray images utilizing transfer learning with convolutional neural networks". In: *Physical and Engineering Sciences in Medicine* 43.2 (2020), pp. 635–640.

[43] Benjamin Q Huynh, Hui Li, and Maryellen L Giger. "Digital mammographic tumor classification using transfer learning from deep convolutional neural networks". In: *Journal of Medical Imaging* 3.3 (2016), p. 034501.

[44] Neophytos Stylianou et al. "Mortality risk prediction in burn injury: Comparison of logistic regression with machine learning approaches". In: *Burns* 41.5 (2015), pp. 925–934.

[45] Abdallah Bashir Musa. "Comparative study on classification performance between support vector machine and logistic regression". In: *International Journal of Machine Learning and Cybernetics* 4.1 (2013), pp. 13–24.

[46] Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. *The elements of statistical learning*. Vol. 1. 10. Springer series in statistics New York, 2001.

[47] Leo Breiman. "Random forests". In: *Machine learning* 45.1 (2001), pp. 5–32.

[48] Ahmad Taher Azar et al. "A random forest classifier for lymph diseases". In: *Computer methods and programs in biomedicine* 113.2 (2014), pp. 465–473.

[49] Philip H Swain and Hans Hauska. "The decision tree classifier: Design and potential". In: *IEEE Transactions on Geoscience Electronics* 15.3 (1977), pp. 142–147.

[50] D Lavanya and K Usha Rani. "Ensemble decision tree classifier for breast cancer data". In: *International Journal of Information Technology Convergence and Services* 2.1 (2012), p. 17.

[51] Leo Breiman. “Bagging predictors”. In: *Machine learning* 24.2 (1996), pp. 123–140.

[52] Dieter Heck et al. “CORSIKA: A Monte Carlo code to simulate extensive air showers”. In: *Report fzka* 6019.11 (1998).

[53] Kosuke Osumi, Takayoshi Yamashita, and Hironobu Fujiyoshi. “Domain Adaptation using a Gradient Reversal Layer with Instance Weighting”. In: *2019 16th International Conference on Machine Vision Applications (MVA)*. IEEE. 2019, pp. 1–5.

[54] Marco Loog. “Nearest neighbor-based importance weighting”. In: *2012 IEEE International Workshop on Machine Learning for Signal Processing*. IEEE. 2012, pp. 1–6.

[55] Steffen Bickel, Michael Brückner, and Tobias Scheffer. “Discriminative learning under covariate shift.” In: *Journal of Machine Learning Research* 10.9 (2009).

[56] Franz Aurenhammer. “Voronoi diagrams—a survey of a fundamental geometric data structure”. In: *ACM Computing Surveys (CSUR)* 23.3 (1991), pp. 345–405.

[57] Rolf Klein. “Voronoi Diagrams and Delaunay Triangulations”. In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. New York, NY: Springer New York, 2016, pp. 2340–2344. ISBN: 978-1-4939-2864-4. DOI: `10.1007/978-1-4939-2864-4_507`. URL: `https://doi.org/10.1007/978-1-4939-2864-4_507`.

[58] Wouter Kouw and Michael Oliver. *Library of transfer learners and domain-adaptive classifiers (libTLDA)*. 2019. URL: `https://github.com/wmkouw/libTLDA` (visited on 03/05/2021).

[59] Michael W Browne. “Cross-validation methods”. In: *Journal of mathematical psychology* 44.1 (2000), pp. 108–132.

[60] Oxford Dictionary. “Overfitting”. In: *Oxford Dictionary*. URL: `https://www.lexico.com/definition/overfitting` (visited on 02/26/2021).

[61] Sylvain Arlot, Alain Celisse, et al. “A survey of cross-validation procedures for model selection”. In: *Statistics surveys* 4 (2010), pp. 40–79.

[62] Seymour Geisser. “The predictive sample reuse method with applications”. In: *Journal of the American statistical Association* 70.350 (1975), pp. 320–328.

[63] Lorenzo Bruzzone and Mattia Marconcini. “Domain adaptation problems: A DASVM classification technique and a circular validation strategy”. In: *IEEE transactions on pattern analysis and machine intelligence* 32.5 (2009), pp. 770–787.

[64] T-P Li and Y-Q Ma. “Analysis methods for results in gamma-ray astronomy”. In: *The Astrophysical Journal* 272 (1983), pp. 317–324.

[65] S Gillessen and HL Harney. “Significance in gamma-ray astronomy–the Li & Ma problem in Bayesian statistics”. In: *Astronomy & Astrophysics* 430.1 (2005), pp. 355–362.