University of Warsaw
Faculty of Physics

Wojciech Noskowiak

Record book number: 417909

Calculations of electric dipole moments in two-electron diatomic molecules

Master's thesis
in the field of Physics

The thesis was written under the supervision of

prof. dr hab. Krzysztof Pachucki
Chair of quantum optics and atomic physics
Institute of Theoretical Physics

mgr Michał Siłkowski
Chair of quantum optics and atomic physics
Institute of Theoretical Physics

Warsaw, September 2022

Summary

An efficient approach to high-precision calculations of expectation values of electronic operators in two-electron diatomic molecules is outlined and explored. A detailed description of the method and its usefulness is provided and illustrated with numerical calculations of the permanent dipole moment and energy of the HeH^+ molecule in its ground state. A comparison of obtained results against values present in the literature is provided and discussed.

Keywords

High-precision calculations, dipole moment, two-electron diatomic molecules, helium hydride ion, Kołos-Wolniewicz, variational method, electronic structure

Title of the thesis in Polish language

Obliczenia elektrycznych momentów dipolowych dla dwuelektronowych dwuatomowych cząsteczek

Contents

Introduction

Neutrinos are fundamental particles that come in three known types (also referred to as flavors). They interact only via the weak nuclear force, making them difficult to detect. In the late 1960s, experiments observed a deficit in the number of solar neutrinos, which was later explained by the phenomenon of neutrino oscillations, which have since been the subject of extensive experimental and theoretical investigation. The mechanism by which neutrinos acquire mass remains unknown, since their masses have not yet been measured.

In addition, several experimental anomalies have been reported that deviate from the predictions of the three-flavor neutrino oscillation model. These include results from the LSND and MiniBooNE experiments, as well as the reactor neutrino anomaly. In all cases, the observed number of neutrinos does not align with the expectations derived from standard oscillation models.

One possible explanation for these discrepancies involves extending the current model by introducing additional neutrino flavors. In particular, the hypothesis of sterile neutrinos—neutrinos that do not interact via the weak nuclear force—has been proposed. If such a hypothesis would be confirmed it would require an extension of the Standard Model of particle physics. These potential non-interactive neutrinos have even been considered as potential candidates for dark matter.

Neutrino oscillations—both standard and non-standard—have so far been studied primarily using water or ice based Cherenkov detectors and scintillation detectors. More recently, liquid argon detectors have started being more commonly deployed.

These detectors offer superior spatial resolution to their predecessors,and efficient analysis and interpretation of data produced by the aforementioned detectors is still an open problem.

This work explores the application of machine learning methods to this problem. Specifically, it investigates and implements selected approaches and evaluates their performance using the publicly available MicroBooNE OpenSamples dataset.

Chapter 1

Neutrinos

1.1. Iterations

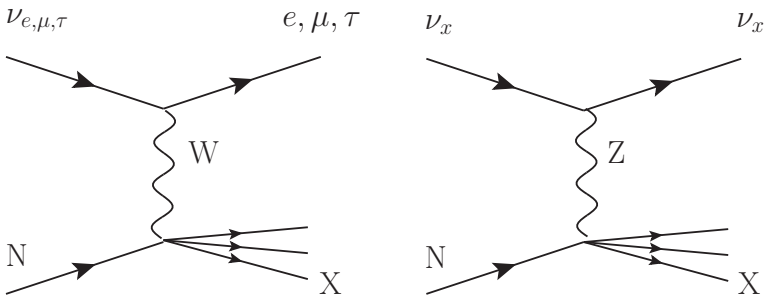

Figure 1.1: Feynman diagram illustrating neutrino interactions.

Neutrinos are electrically neutral, spin-$\frac{1}{2}$ elementary particles that belong to the lepton family within the Standard Model of particle physics. They exist in three distinct flavors, corresponding to the three charged leptons: the electron neutrino (ν_e), the muon neutrino (ν_μ), and the tau neutrino (ν_τ).

These flavor classifications arise from the weak decay processes mediated by the W boson, in which a charged lepton is produced alongside a corresponding neutrino. For instance, in a W^- decay producing an electron, an electron neutrino is emitted; similarly, decays yielding a muon or tau lepton produce a muon or tau neutrino, respectively. Feynman diagrams of these interactions are shown in figure 1.1.

Due to their lack of electric charge, negligible mass, and absence of color charge, neutrinos interact with matter only via the weak nuclear force. This interaction occurs through two fundamental mechanisms: neutral current interactions, mediated by the electrically neutral Z^0 boson, and charged current interactions, mediated by the charged W^+ or W^- bosons.

In charged current interactions, the neutrino exchanges a W boson with a target particle, resulting in the production of a charged lepton of the same flavor, which enables neutrino flavor identification. In contrast, neutral current interactions involve the exchange of a Z^0 boson and leave the neutrino unchanged, making the final state flavor-indistinguishable.

The weak interaction is characterized by a very short range, typically on the order of 10^{-17} to 10^{-18} meters, which makes neutrino interactions with matter extremely rare. As a result, detecting neutrinos requires either very intense neutrino beams and/or very large detectors to observe a statistically significant number of interactions.

1.2. Liquid Argon Detectors

1.2.1. Operating principle

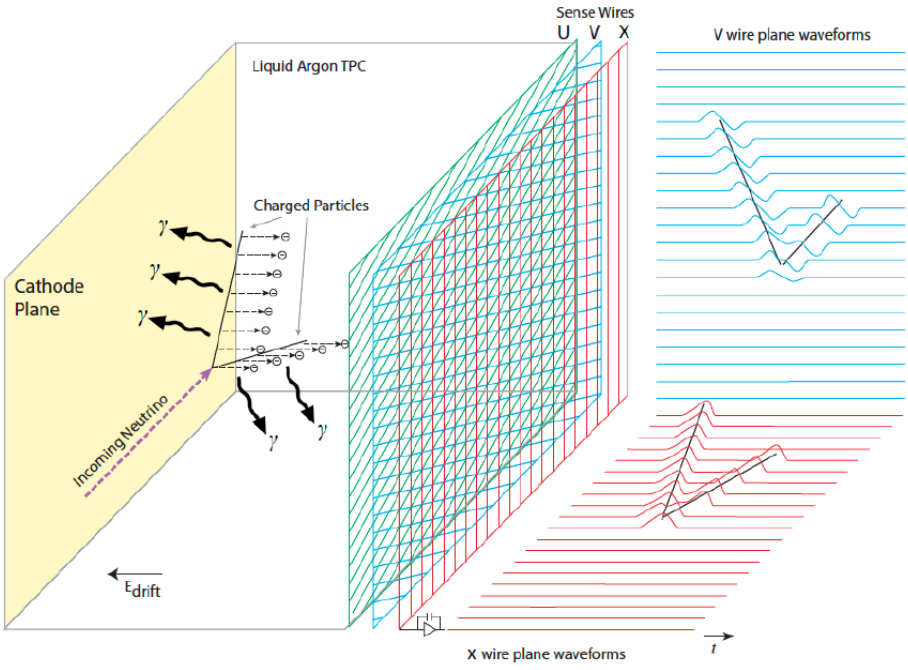

Figure 1.2: General mechanism of a liquid argon detector (LArTPC).

LArTPC detectors consist of a large chamber filled with liquid argon. Charged particles traversing the chamber leave trails of ionized electrons. An applied electric field causes these electrons to drift toward the anode, where they are detected. Simultaneously, vacuum ultraviolet (VUV) scintillation photons are emitted and collected by photodetectors. The general mechanism of a liquid argon detector is shown in Figure 1.2.

Argon is chosen as the detection medium because it is a noble gas, which is chemically

inert and allows for efficient ionization and scintillation. In its liquid state, argon has a high density, enhancing the probability of neutrino interactions. Importantly, argon does not capture ionization electrons, allowing them to drift over long distances without recombination—provided the medium is pure.

When a neutrino interacts with an argon nucleus, it produces charged particles that traverse the medium, leaving behind trails of ionized electrons. The detector volume is subjected to a uniform electric field, which causes these free electrons to drift toward a set of anode planes, where their arrival is recorded. At the same time, the interaction generates VUV scintillation photons, which are detected by an array of photodetectors placed around the chamber. The information from the ionization charge provides two spatial coordinates, while the timing information from the scintillation light gives the third coordinate, enabling full three-dimensional reconstruction of particle trajectories and interaction vertices within the detector volume.

1.2.2. Noise

Figure 1.3: Example of noise patterns in lartcp detectors, taken from microBooNe

Neutrino interactions are not the only source of signals in the detector. Cosmic muons produced by secondary cosmic rays in the atmosphere can also ionize the detector medium, producing characteristic noise patterns. These patterns are typically distinct and easily distinguishable from genuine neutrino events. They often take the form of long, low-intensity tracks oriented in a single direction 1.3. The rate of such background events strongly depends on how deep the detector's been placed underground. For this reason, LArTPC detectors are usually placed at significant depths. However, complete elimination of these events is not possible. Therefore such events need to be accounted for and filtered out during data processing.

Chapter 2

MicroBooNE Experiment

2.1. Detector

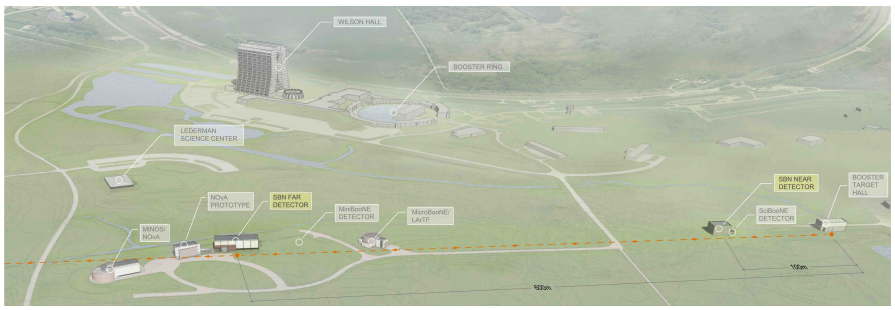

Figure 2.1: Aerial diagram showing location of MicroBooNE.

The MicroBooNE experiment began operation at Fermilab in 2015 and currently functions as the near detector in the Short-Baseline Neutrino (SBN) program. It's primary scientific goal is to investigate the source of the excess of electron neutrino-like events observed in the predominantly muon neutrino beam by the MiniBooNE experiment. It's exact placement in the BNB (Booster Neutrino Beam) is shown in Figure 2.1.

MicroBooNE is based on a 170-ton liquid-argon TPC, enclosed in a cryostat and depicted in Figure 2.2. It's general operating principal is strictly analogous to the one described by 1.2 The detector is located on the Fermilab site, as shown in Figure 2.1. Key design parameters are summarized in Table 2.1.

In addition to its primary scientific goals, it contributed essential input toward the development of kiloton-scale liquid-argon time projection chambers (LArTPCs) for the upcoming Deep Underground Neutrino Experiment (DUNE).

Table 2.1: Primary detector design parameters for MicroBooNE.

Parameter	**Value**
LArTPC Dimensions	2.325 m vertically 2.560 m horizontally 10.368 m longitudinally
LArTPC argon mass	90 tonnes
Total Number of Wires	8256
Drift field	500 V/cm
Light collection	32 200 mm (8 in) diameter PMTs 4 lightguide paddles
Total liquid argon mass	170 tonnes
Operating temperature	87 K
Depth	1.5 m

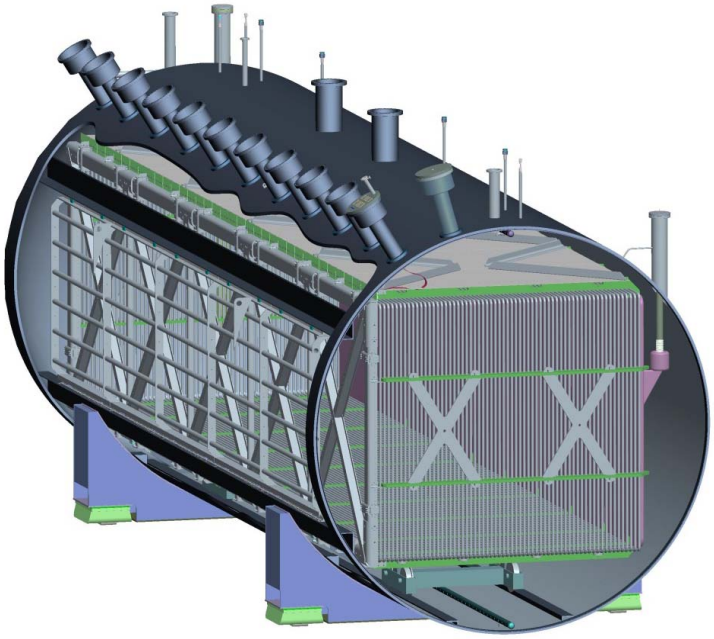

Figure 2.2: Schematic diagram of the MicroBooNE LArTPC inside the cryostat.

2.1.1. Publicly available datasets

Two MicroBooNE datasets are opened to the public. They contain simulated neutrino interactions overlaid on top of cosmic ray data. These datasets were provided to support the development of new computational techniques for neutrino event reconstruction and analysis, primarily with DUNE in mind. They will be discussed in greater detail in the next chapter.

Chapter 3

MicroBooNE Public Datasets

All the data used further in this work are two publicly available datasets from the MicroBooNE experiment: the Inclusive sample and the Electron Neutrino sample. The Inclusive sample contains both charged-current (CC) and neutral-current (NC) interactions, as well as various neutrino types, reflecting the true composition of the Booster Neutrino Beam. The Electron Neutrino sample, on the other hand, contains only electron neutrino CC interactions.

The MicroBooNE OpenSamples dataset is provided in two formats: HDF5 and art/ROOT. The HDF5 format is a hierarchical data format suitable for efficient storage and access to large numerical datasets. The art/ROOT format is based on the ROOT framework, which is commonly used in high-energy physics for storing and analyzing complex event data structures [**?**, **?**].

There are two main types of data samples: "with wires" and "no wires." The "with wires" datasets include full waveform data from the detector wires, allowing for signal reconstruction and image generation. The "no wires" datasets omit this raw waveform information and therefore do not allow for image reconstruction.

3.1. Events Present

3.1.1. BNB sample

The distributions of lepton and neutrino energies for the two datasets are shown in Figures 3.1–3.4.

The BNB (Booster Neutrino Beam, Inclusive) sample contains a total of 24,332 events, all of which are classified as charged-current (CC) interactions. The dataset is composed exclusively of electron neutrinos and antineutrinos, with 19,448 events corresponding to neutrinos (PDG code 12) and 492 to antineutrinos (PDG code -12). No muon neutrinos (PDG 14 or -14) are present.

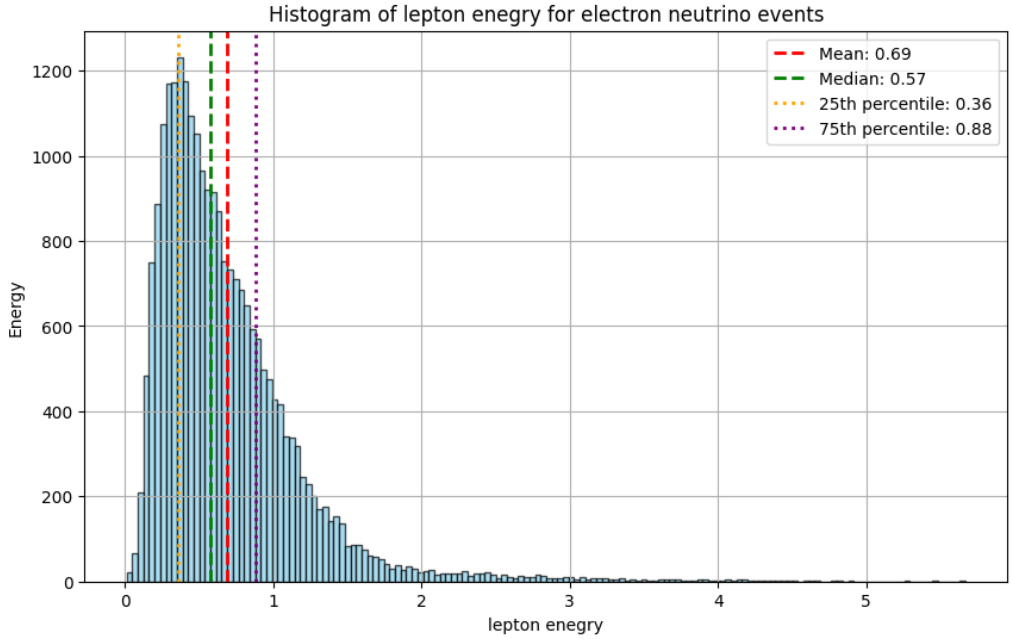

Figure 3.1: Lepton energy distribution in the Inclusive (BNB) sample. The vertical axis shows the number of events.

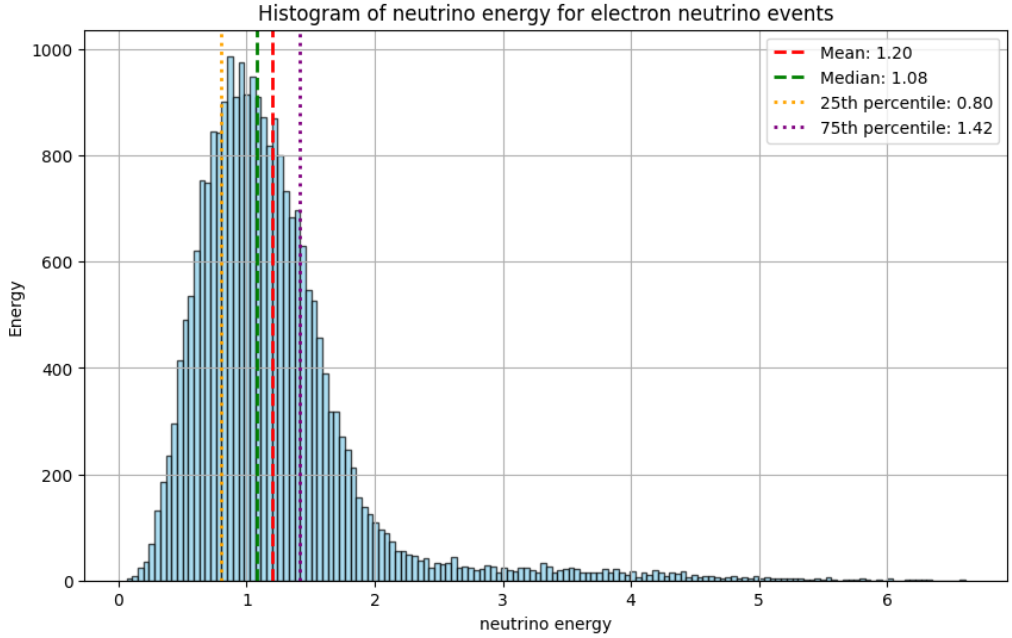

Figure 3.2: Neutrino energy distribution in the Inclusive (BNB) sample. The vertical axis shows the number of events.

The energy distribution of neutrino interactions in this sample has a mean value of 1.62 GeV, a median of 1.46 GeV, and a standard deviation of 0.86 GeV. The minimum recorded energy is 0.13 GeV, while the maximum reaches 6.13 GeV.

The associated charged lepton energy distribution exhibits a mean of 0.69 GeV, a median of 0.57 GeV, and a standard deviation of 0.50 GeV. The lepton energies range from 0.01 GeV to 5.67 GeV. These characteristics reflect a relatively high-energy subset of interactions, with lepton energy distributions concentrated in the sub-GeV to low-GeV range.

3.1.2. ν_e-Enhanced Sample Description

The distributions for the ν_e-enhanced sample are shown in Figures 3.3 and 3.4.

The ν_e-enhanced sample consists of 19,940 events, containing a broader mix of neutrino types. The majority are muon neutrinos (23,984), followed by smaller numbers of muon antineutrinos (208 with PDG code -14), electron neutrinos (134 with PDG code 12), and electron antineutrinos (6 with PDG code -12). Of the total, 17,564 events are classified as charged-current (CC), while 6,768 are neutral-current (NC) interactions. Within the charged-current subset, 17,457 events originate from muon neutrinos or antineutrinos, and 107 from electron neutrinos or antineutrinos, indicating a dominance of muon neutrino interactions in this sample.

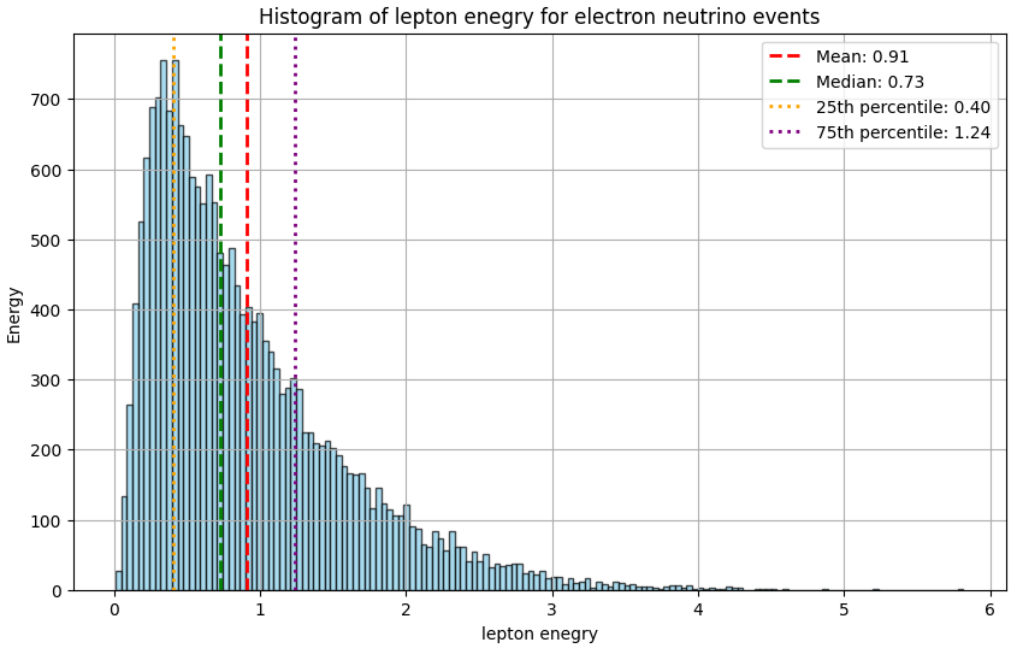

Figure 3.3: Lepton energy distribution in the nu_e-enhanced sample. The vertical axis shows the number of events.

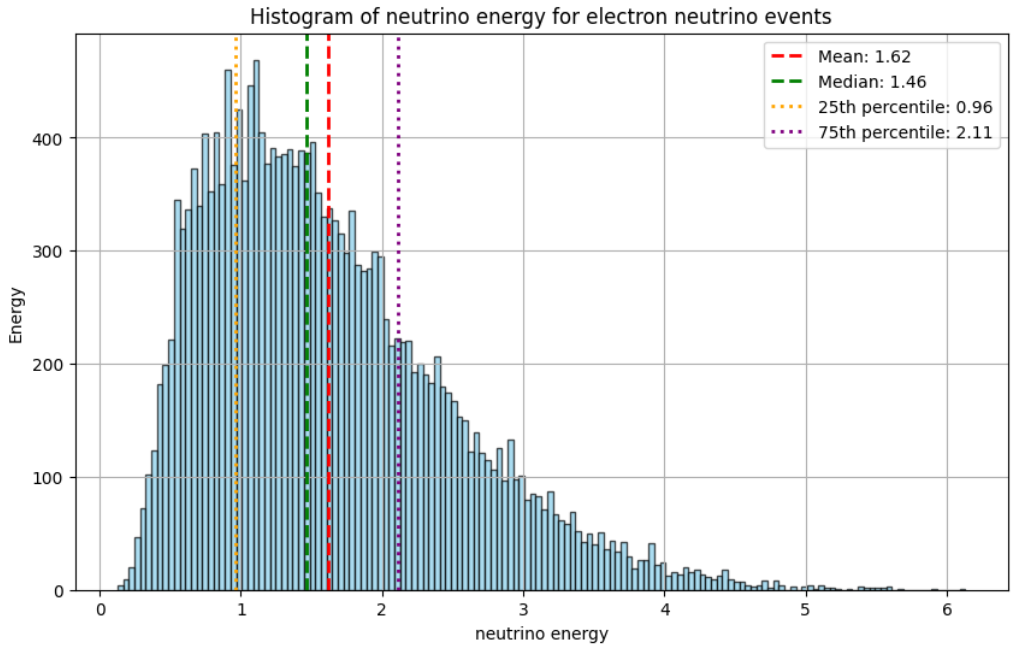

Figure 3.4: Neutrino energy distribution in the nu_e-enhanced sample. The vertical axis shows the number of events.

The energy distribution shows a mean of 1.20 GeV, a median of 1.08 GeV, and a standard deviation of 0.67 GeV, with values ranging from 0.06 GeV to 6.61 GeV. The corresponding charged lepton energy distribution has a mean of 0.91 GeV, a median of 0.73 GeV, and a standard deviation of 0.67 GeV. Lepton energies range from 0.01 GeV to 5.82 GeV. This distribution suggests a slightly lower-energy spectrum on average, but with a broader mix of interaction types and moderate lepton energy variability.

3.1.3. Combined Dataset

The combined dataset consists of 44,272 events and includes a mix of neutrino interaction types. The dominant component is muon neutrinos (23,984), followed by electron neutrinos (19,582), electron antineutrinos (498), and a small number of muon antineutrinos (208). Out of the total, 37,504 events are classified as charged-current (CC) interactions, while 6,768 are neutral-current (NC). Within the CC category, 17,457 events originate from muon neutrinos or antineutrinos, and 20,047 from electron neutrinos or antineutrinos, indicating a more balanced representation of flavor types compared to the ν_e-enhanced sample. The charged lepton energy distribution exhibits a mean of 0.79 GeV, a median of 0.63 GeV, and a standard deviation of 0.60 GeV. The energy values range from 0.01 GeV to 5.82 GeV. This suggests a lower average energy compared to the ν_e-enhanced sample, with a more symmetric distribution and substantial contributions from both muon and electron neutrino interactions.

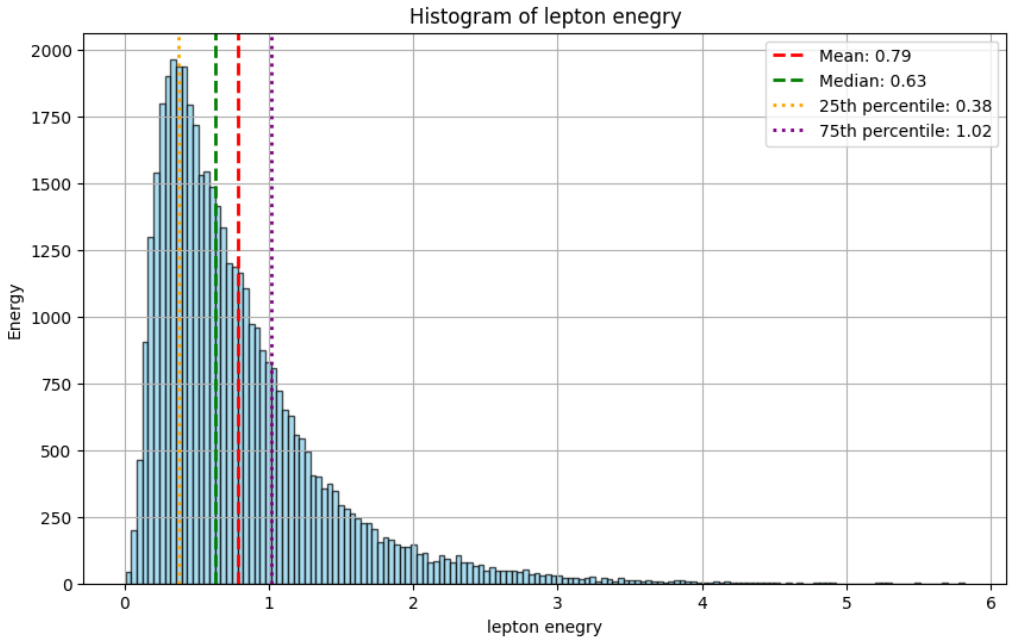

Figure 3.5: Lepton energy distribution in the combined dataset (Inclusive + nu_e-enhanced). The vertical axis shows the number of events [count].

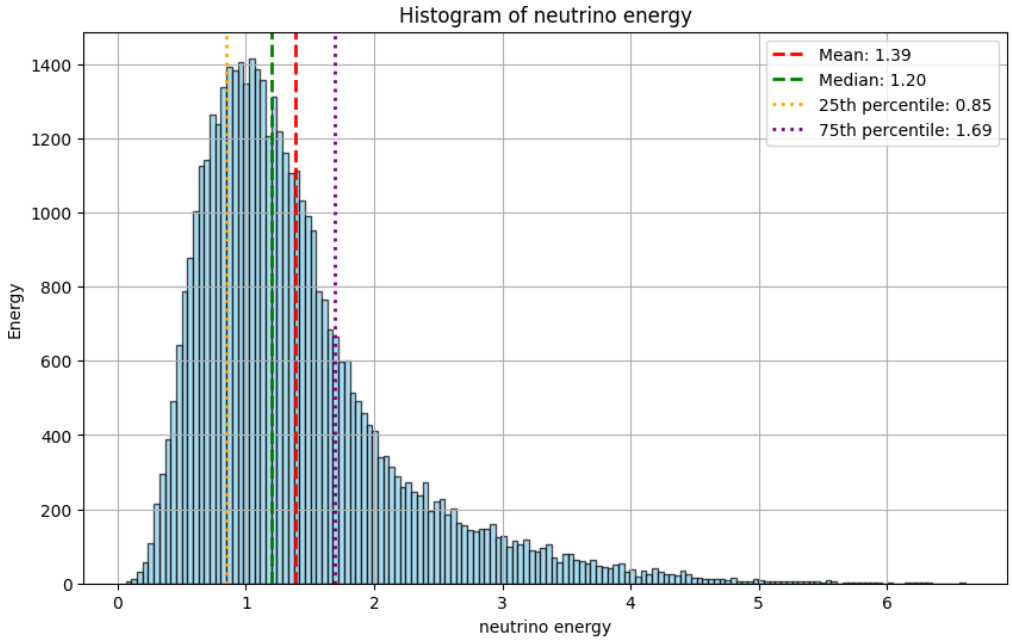

Figure 3.6: Neutrino energy distribution in the combined dataset (Inclusive + nu_e-enhanced). The vertical axis shows the number of events [count].

Table 3.1: Summary of Neutrino Events in the Dataset

Category	**Count**
Total events	44,272
BNB sample	24,332
ν_e-enhanced sample	19,940
In ν_e-enhanced sample	
ν_μ (PDG = 14)	23,984
$\bar{\nu}_\mu$ (PDG = -14)	208
ν_e (PDG = 12)	134
$\bar{\nu}_e$ (PDG = -12)	6
Charged current (CC) events	17,564
Neutral current (NC) events	6,768
CC ν_e or $\bar{\nu}_e$ events	107
CC ν_e events	101
CC ν_μ or $\bar{\nu}_\mu$ events	17,457
CC ν_μ events	17,338
Energy statistics (ν_e-enhanced sample)	
Mean (GeV)	1.20
Median (GeV)	1.08
Std. deviation (GeV)	0.67
Min (GeV)	0.06
Max (GeV)	6.61
In BNB sample	
ν_e (PDG = 12)	19,448
$\bar{\nu}_e$ (PDG = -12)	492
CC events	19,940
CC ν_e or $\bar{\nu}_e$ events	19,940
CC ν_e events	19,448
CC ν_μ or $\bar{\nu}_\mu$ events	0
Energy statistics (BNB sample)	
Mean (GeV)	1.62
Median (GeV)	1.46
Std. deviation (GeV)	0.86
Min (GeV)	0.13
Max (GeV)	6.13

Chapter 4

Dataset preparation

The Jupyter notebooks accompanying the MicroBooNE OpenSamples repository recommend creating a Conda environment that includes the necessary packages. While there is nothing inherently wrong with this approach, it does introduce several disadvantages. Beyond general inefficiencies—such as longer environment loading times, higher memory usage, and increased disk space consumption—the most significant issue encountered was related to the sheer number of files generated by Conda distributions.

Both Conda and Miniconda installations can create and maintain hundreds of thousands of files. Although the total disk usage of these files is typically not problematic (many are symbolic links or small metadata files), the system used for processing data enforced strict quotas on the number of files each user could create. The data processing was performed on the ICM UW "Okeanos" cluster running CentOS 7. Installing a full Conda distribution brought the user account dangerously close to this limit, which severely impacted the usability of the system. For instance, launching a new Jupyter session became almost impossible, as nearly every action related to managing Jupyter generates additional log files, further pushing against the file quota.

Additionally, the environment setup proposed in the MicroBooNE OpenSamples notebooks depended on outdated libraries—most notably `pynuml` version 0.3, which is no longer maintained and is poorly documented. This package also requires an outdated version of Python (3.7) and depends on a deprecated version of HDF5. These constraints make it effectively impossible to use current versions of many essential packages, which significantly complicates the development process.

To address these issues and improve the portability and maintainability of the tools used for processing OpenSamples data, an alternative approach to environment management was adopted.

4.0.1. Containerization and Singularity

Containerization is a method of packaging software and its dependencies into isolated, self-contained units called containers. Each container includes all necessary binaries, libraries, and configuration files, ensuring that applications run consistently across different systems regardless of variations in the underlying infrastructure. Modern containerization solutions allow for building, distributing, and executing containers in a lightweight and reproducible manner. Containers are typically defined using a configuration file that builds on a base image and specifies additional dependencies and setup steps, producing a new image that can be versioned and reused.

In this specific case, a containerization tool called Singularity was used. Singularity is particularly well-suited for high-performance computing environments, as it enables users to define, build, and execute containers without requiring root privileges. A container definition file was created based on a Ubuntu 22.01 base image. (An image is a static, read-only template with the application and its dependencies, while a container is a running instance of an image, including its own filesystem and processes.) This definition specified the installation of all required dependencies and packages during the container build process. Using this approach made it possible to work with up-to-date software versions, including Python 3.9, pynuml version 25.4.0, and HDF5 version 1.12.2, although these upgrades required some code refactorings to be performed on the code present in OpenSamples notebooks. Moreover, since the resulting container image is stored as a single file, this method also effectively circumvented the file quota limitations encountered with Conda-based setups.

4.0.2. HDF5 File Structure

Data stored in HDF5 files was separated into multiple tables, each containing a specific type of information. The tables are organized hierarchically, with the root directory containing the main event table and additional subdirectories for other data types. Each table is structured as a collection of records, where each record corresponds to a single event or hit in the detector. The tables are designed to be self-describing, meaning that they include metadata about the data types and structures used within them. The HDF5 file structure for the MicroBooNE OpenSamples dataset contains multiple tables with specific information types. A detailed description of all tables is provided in Appendix .1.

Because of their smaller size, HDF5 files were used for the initial data extraction. Although the HDF5 format is well-suited for storing large datasets, reading from it can require loading substantial portions of data into memory—a time-consuming operation that makes it impractical to use these files directly for neural network training. Therefore, a decision was made to extract the relevant data and store it in a simpler, more accessible format.

The final extracted dataset consisted of a CSV file containing the composite event ID (run number, subrun number, and event number), the lepton energy, neutrino energy, and the `is_cc` value from the event table, as well as the `PDG` of the interacting neutrino.

Additionally, it included the filenames of three image files, each corresponding to one of the three wire planes. These images were generated from the wire readouts and represent visualizations of the recorded signals.

4.0.3. Data extraction

Data from the `wire_table`, `edep_table`, and `lep_table` was first matched with corresponding events in the `event_table` (this was largely based on the code present in the Wireimage.ipynb notebook provided by the authors of the dataset). From each matched record, the lepton energy was extracted from the `lep_table`, the neutrino energy from the `edep_table`, and the `is_cc` value from the `event_table`. The `wire_table` was used to obtain the wire signals for each event. It was subsequently processed into images, and linked to the corresponding event metadata for downstream analysis.

4.1. Image Rendering

Figure 4.1: Example of a wire readout image

Images were generated using data from the wire readouts, where the horizontal axis corresponds to the wire number and the vertical axis to time. Each pixel represents the signal recorded on a specific wire at a given time. To suppress noise, a threshold was applied, below which wire activity was not visualized. Additionally, a saturation limit was introduced to prevent high-intensity signals from dominating the image. Pixel intensities were linearly scaled to an 8-bit range (0–255), and the resulting images were downsampled by a factor of two to reduce dataset size. This procedure was applied to all three wire planes, producing three images per event. These images were later saved as PNG files, and their filenames were stored in an appropriate CSV manifest file (a summary file listing all images and their associated metadata for each event).

4.1.1. Final Dataset

The final dataset consisted 132,816 PNG images, each representing a wire readout from the MicroBooNE detector alongside 38 CSV manifest files, each representing events form

a single HDF5 file. This dataset turend out to contain approximately 13 GB of data, with each image file averaging around 100 KB in size.

4.2. Further Dataset Processing

4.2.1. Sparse array representation

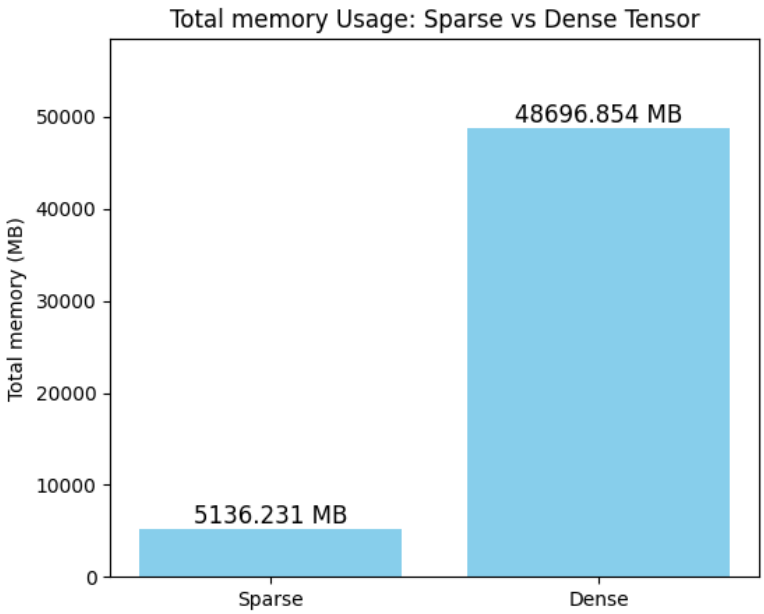

Figure 4.2: Comparison of total memory usage between dense and sparse array representations.

Although the final image dataset occupied approximately 13 GB on disk, this was primarily due to the use of compressed image formats (specifically, PNG). When these images were loaded into memory as arrays of integers, their size increased dramatically—by more than a factor of 100 in some cases. It quickly became evident that the dataset was too large to be feasibly loaded into the system's RAM without causing memory fragmentation. However, loading each image individually into memory incurred significant computational overhead: each image had to be read from disk, memory had to be allocated, and the image had to be decompressed before being converted into an array. Since training a convolutional neural network (CNN) typically requires each image to be loaded multiple times, it was crucial to minimize the number of image loading operations.

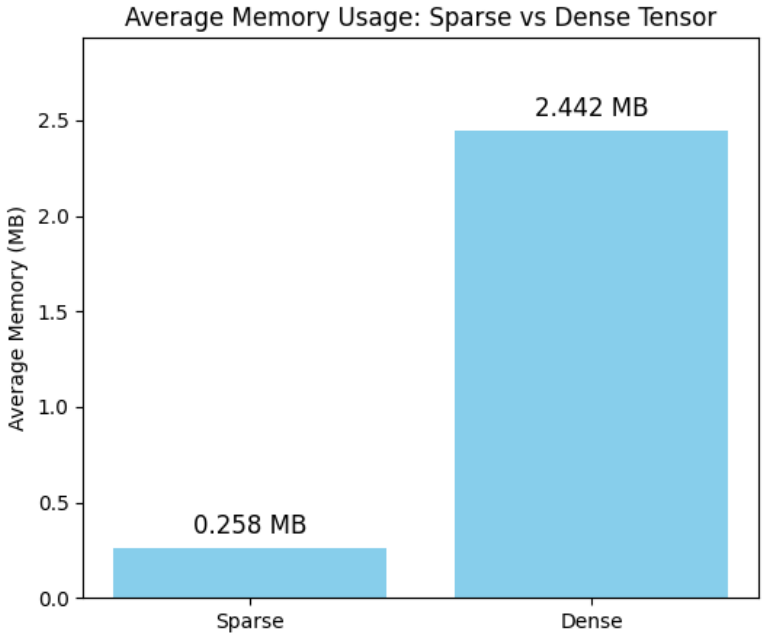

Figure 4.3: Comparison of average memory usage between dense and sparse array representations.

During development, it was observed that the processed images contained mostly empty (zero-value) pixels. While tracks were present in all images, they typically occupied only a small portion of each image—approximately 5% of the total pixels. (Note: Mention that the threshold used during the generation of wire images was sufficiently low to eliminate noise.) As a result, the decision was made to represent the images using sparse arrays (implemented with `torch.sparse`) Unlike dense arrays, where each element is stored explicitly in memory, sparse arrays assume a default value for all positions and store only the non-zero values along with their coordinates. Although these sparse arrays still need to be converted to dense format before being passed to the CNN, this conversion is significantly less costly than loading and decompressing images from disk. This optimization led to a substantial reduction in memory usage—by up to a factor of 70—making it feasible to load the entire dataset into RAM. The magnitude of those size reductions are shown in Figures 4.2 and 4.3.

4.2.2. File count reduction

The final dataset contained a significant amount of files - namely approximately 120,000 image files. This created issues similar to those previously encountered when working with Miniconda—namely, the operaing system configurations used on ICM rysy and topola clusters used for CNN training imposed quotas on the number of files each user could allocate. Operating under an account close to that limit proved to be cumbersome.

Notably, the total dataset size remained well below the storage quota in terms of bytes, indicating that reducing the number of files could resolve the problem without affecting overall storage usage.

To address both issues, each CSV file was loaded into a pandas DataFrame. Three additional columns were added to each record—one for each image associated with an event—and populated with the corresponding sparse arrays. Each resulting dataset was then saved as a single `.pkl` file, a format that preserves Python objects as they exist in memory. In total, 38 .pkl files were created, each corresponding to one original HDF5 file and containing all events and associated images from that file. The combined size of all `.pkl` files was approximately 32 GB, which corresponds closely to the amount of memory required to load them into RAM.

Chapter 5

Classification Task definition

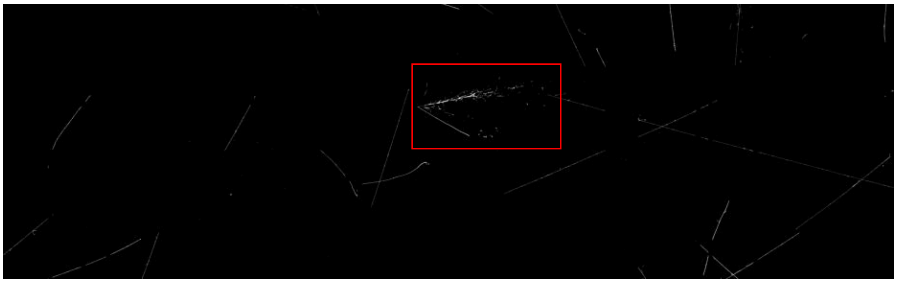

Figure 5.1: Example of an electromagnetic shower produced by an electron neutrino interaction in a LArTPC detector (marked in red).

A general heuristic can be assumed: the more significant the visual difference between two classes in a classification problem, the better an image classification algorithm will perform. However, the problem should remain non-trivial and possess potential scientific utility.

As mentioned before, charged current neutrino interactions produce leptons. The different types of leptons resulting from these interactions can be distinguished. When passing through a LArTPC chamber, electrons—produced in electron neutrino interactions—generate electromagnetic showers. These showers are highly visible and have a very distinguishable shape (they appear as 'showers', not tracks), making them easily identifiable 5.1.

For these reasons, after some deliberation, a decision was made to divide the dataset based on whether or not a charged current electron neutrino interaction occurred in a given sample.

To simplify the problem further (by narrowing the data down to interactions with the most pronounced electron showers), an additional dataset was created. This subset

contains only the 50% most energetic electron neutrino charged current interactions. The size of the other class was also reduced arbitrarily to avoid class imbalance. This dataset is further referred to in the work as `top50`.

All the machine learning models discussed further in this work were trained on these aforementioned classification tasks. A summary of the number of members of each class for each dataset can be found in Table 5.1.

Table 5.1: Summary statistics of the full and `top50` datasets used for classification.

Dataset	**Total Samples**	**Other**	**CC μ_e**
Full Dataset	44,272	24,225	20,047
`top50` Dataset	22,136	10,739	11,397

Chapter 6

Machine learning fundamentals

Machine learning algorithms aim to find a mapping from a specific set of points in an input space to a corresponding set of points in an output space, both of which are predefined. These algorithms define a procedure for learning such a mapping based on a given set of input-output pairs, commonly referred to as the training data. Once learned, this mapping can be applied to all points in the input space, including those not included in the training set. The general expectation is that the mapping learned from the training data will generalize well to previously unseen data, enabling the algorithm to make accurate predictions or classifications on new inputs.

6.1. Supervised machine learning

The problems tackled in this work are classification problems, where the input space consists of images of wire readouts, and the output space consists of a binary label. The common way of tackling such problems is supervised machine learning. In supervised machine learning, the training data is given in the form of input-target pairs $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$, where $\mathbf{x}_i \in \mathbb{R}^d$ represents the feature vector of the i-th input and $y_i \in \mathcal{Y}$ is the corresponding target value. The aim is to learn a function $f : \mathbb{R}^d \to \mathcal{Y}$ that approximates the true relationship between inputs and outputs.

The central objective is to find a function f such that:

$$f(\mathbf{x}_i) \approx y_i \quad \forall i \in \{1, \ldots, N\}$$

6.2. Machine Learning Models

6.2.1. ML model definition and learnable Parameters

The term machine learning model mentioned further in the work will refer to a pairing of a set of learnable parameters $\boldsymbol{\theta} \in \mathbb{R}^p$, where p is the total number of parameters, and

a model function $f(\mathbf{x}; \boldsymbol{\theta})$ which maps the input space to the output space. The learnable parameters $\boldsymbol{\theta}$ are variables within the model that can be adjusted during the training process to minimize the difference between the model's predictions and the actual target values. They have a direct influence on the model's output.

The training process begins by setting all elements of $\boldsymbol{\theta}$ with some initial values. For each training iteration, a batch of input data $\mathbf{X} \in \mathbb{R}^{b \times d}$ is passed through the model, and the corresponding predictions $\hat{\mathbf{y}} \in \mathbb{R}^b$ are obtained:

$$\hat{\mathbf{y}} = f(\mathbf{X}; \boldsymbol{\theta})$$

These predictions are then compared to the true labels $\mathbf{y} \in \mathbb{R}^b$, and the parameters $\boldsymbol{\theta}$ are updated to reduce the discrepancy. This step is repeated iteratively, refining $\boldsymbol{\theta}$ over time to improve predictive performance. Each such update constitutes a learning step, and the entire iterative process is referred to as model training.

6.2.2. Neural networks

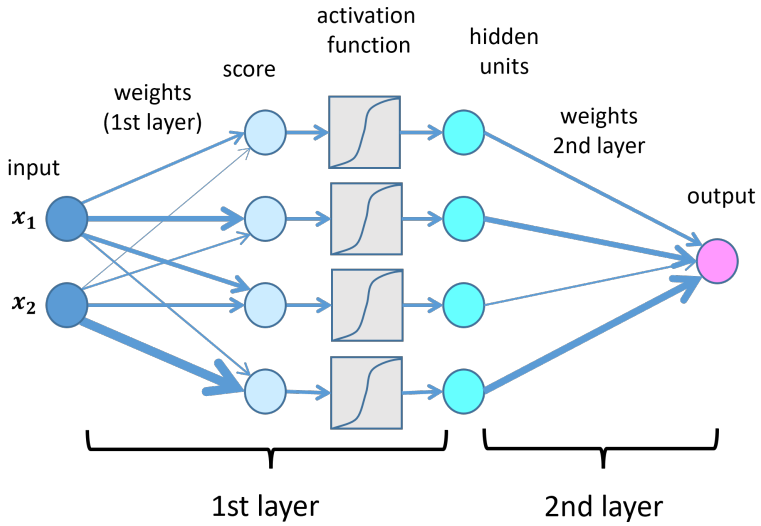

Figure 6.1: Visualization of a neural network

Neural networks are a class of machine learning models inspired by the structure of biological neurons. They consist of layers of interconnected nodes, also known as neurons, where each node computes a weighted sum of its inputs followed by a non-linear transformation. These networks are particularly effective for learning high-dimensional patterns in data, especially in tasks such as image recognition, classification, and regression.

A typical neural network consists of an input layer, one or more hidden layers, and an output layer 6.1. Each layer performs a transformation of the form:

$$\mathbf{h}^{(l)} = g^{(l)}(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)})$$

where $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are the learnable weights and biases of layer l, $g^{(l)}$ is a non-linear activation function, and $\mathbf{h}^{(l-1)}$ is the output of the previous layer.

Activation Functions

Activation functions introduce non-linearity into the model, allowing the network to learn complex and non-linear decision boundaries. Two activation functions are used in the models discussed in this work: ReLU and Softmax.

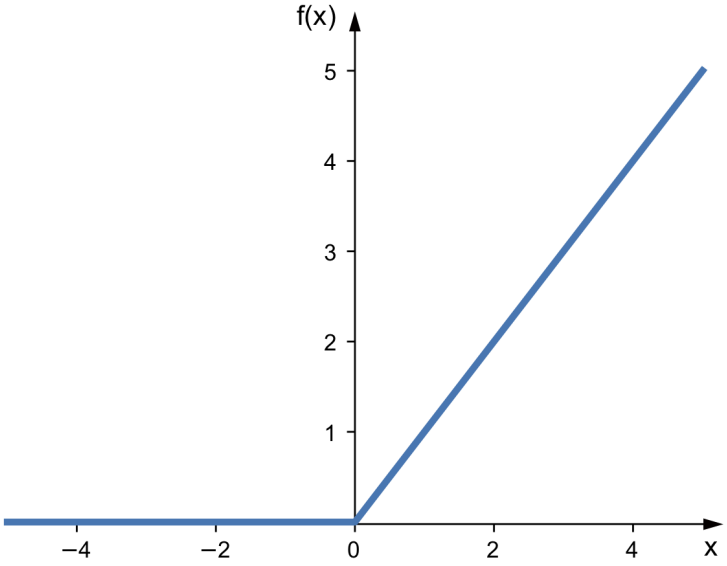

Figure 6.2: Graph of the ReLU activation function.

ReLU The Rectified Linear Unit (ReLU) is a widely used activation function defined as:

$$g(x) = \max(0, x)$$

Example plot of such function is shown in figure 6.2.

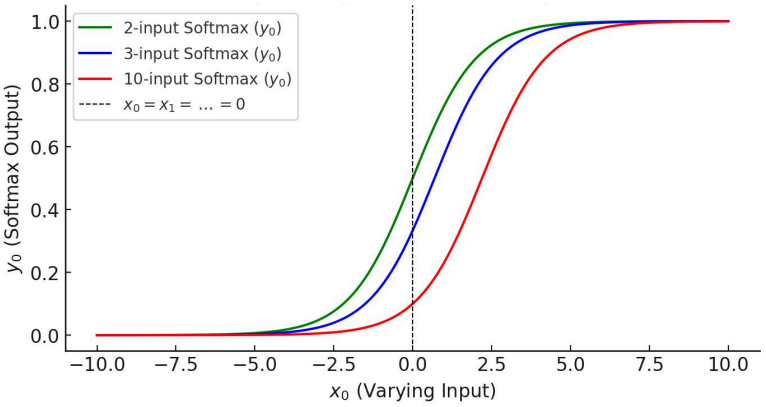

Figure 6.3: Softmax output for y_0 with other inputs fixed at 0.

Softmax The Softmax function is used in the output layer of classification networks to convert raw output scores into a normalized probability distribution over C classes:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}} \quad \text{for } i = 1, \ldots, C$$

This property makes it well-suited for multi-class classification tasks, where each input must be assigned a single label. It's general behaviour can be seen in figure 6.3.

6.2.3. Loss Function

The quality of the model's predictions is assessed using a loss function

$$\mathcal{L}(\hat{y}, y)$$

Here y stands for the true target value, while $\hat{y} = f(\mathbf{x}; \boldsymbol{\theta})$ is the model's prediction for a given input $\mathbf{x}$.

The role of this function is to quantify the difference between predicted and true outputs. By assumtion the value of the loss function is non-negative, with lower values indicating better model performance (the outputs of the model are given).

The choice of an optimal loss function is problem-specific but, regardless of the choice, the goal is to minimize the loss function with respect to the model parameters:

$$\min_{\boldsymbol{\theta}} \; \mathcal{L}(f(\mathbf{x}; \boldsymbol{\theta}), y)$$

Cross-Entropy Loss

Because of the nature of the problems discussed in this work, the cross-entropy loss function was used. This loss function is commonly applied to classification tasks.

The function in question measures the dissimilarity between the predicted probability distribution $\hat{\mathbf{y}}$ over classes and the true class label distribution $\mathbf{y}$, typically represented as a *one-hot encoded vector*—a binary vector of length C, where all entries are zero except for a one at the index corresponding to the correct class. For a single example and C classes, the cross-entropy loss is defined as:

$$\mathcal{L}_{\text{CE}}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

This loss penalizes incorrect confident predictions more heavily and encourages the model to assign high probability to the correct class. Minimizing the cross-entropy therefore drives the model to output probability distributions that closely align with the true labels.

6.2.4. Gradient Descent Optimization

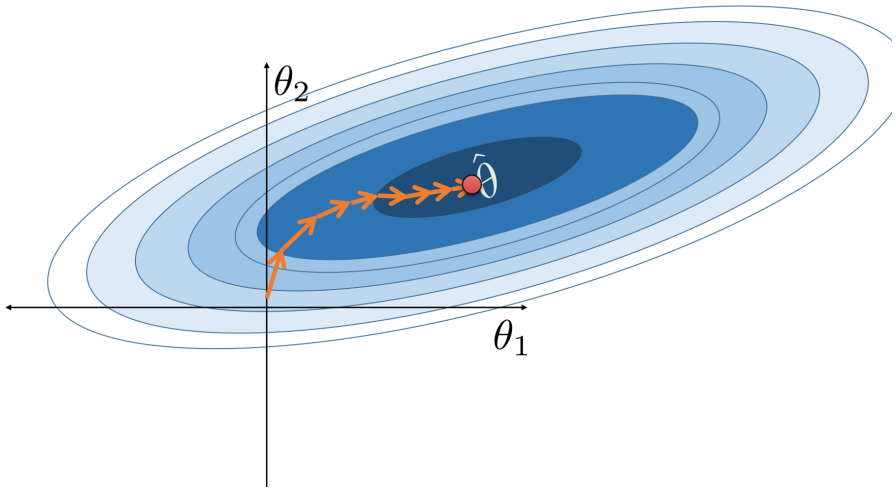

Figure 6.4: Gradient descent visualization. A local minimum is achieved by taking steps of a predefined size in the direction of the steepest descent.

The approach used for training models used in this work is called *gradient descent*. In particular, stochastic gradient descent (SGD). In each iteration, the gradient of the loss function with respect to the model parameters is computed:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial \theta_1}, \frac{\partial \mathcal{L}}{\partial \theta_2}, \ldots, \frac{\partial \mathcal{L}}{\partial \theta_p} \right]^\top$$

This gradient vector indicates the direction in parameter space that increases the loss most steeply. The parameters are then updated in the opposite direction of the gradient to reduce the loss:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}$$

where $\eta > 0$ is the learning rate, a hyperparameter that controls the step size.

By repeating this process over multiple iterations and batches of training data, the parameters ideally converge to a local minimum of the loss function. At convergence, the model's predictions are as close as possible to the target values within the expressiveness of the chosen architecture.

This iterative process of computing gradients and updating parameters is known as *gradient descent*.

Momentum

In order to smooth out the gradient descent and accelerate convergence a *momentum* can be introduced to a sdg algorithm. When introduced the descent direction incorporates a fraction of the previous update step, instead of relying solely on the current gradient.

The momentum update rule is defined as:

$$\begin{aligned} \mathbf{v}_t &= \gamma \mathbf{v}_{t-1} + \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_{t-1}) \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} - \mathbf{v}_t \end{aligned}$$

Here, $\mathbf{v}_t$ is the velocity vector (accumulated update), $\gamma \in [0, 1)$ is the momentum coefficient, and η is the learning rate. A higher γ allows more influence from previous gradients, leading to smoother and potentially faster convergence.

6.2.5. Output Layer and Thresholding

In binary classification tasks, the output layer of the neural network perceptron contains two neurons, each representing one of the two possible classes. The network produces a pair of output values, which are interpreted as unnormalized logit scores (i.e., the raw, unscaled values output). These scores are typically passed through a softmax function to produce a probability distribution over the two classes:

$$\hat{y}_i = \frac{e^{z_i}}{e^{z_0} + e^{z_1}}, \quad i \in \{0, 1\}$$

where z_0 and z_1 are the logits for each class. The final classification decision is made by selecting the class with the higher probability. Alternatively, a fixed threshold t can be applied to the output of a single neuron (if using a one-neuron output with a sigmoid activation) to make the decision:

$$\text{Class} = \begin{cases} 1 & \text{if } \hat{y} > \text{t} \\ 0 & \text{otherwise} \end{cases}$$

Networks discussed further in this work will use a threshold of 0.5.

6.3. Performance Metrics

To evaluate the performance of the trained classification models, several metrics were used. These metrics provide complementary insights into different aspects of classification performance, especially in the context of imbalanced data.

6.3.1. F1 Score

The F1 score is the harmonic mean of precision and recall, offering a balanced measure that accounts for both false positives and false negatives. For a binary classification problem, precision and recall are defined as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

where TP, FP, and FN denote true positives, false positives, and false negatives, respectively. The F1 score is then given by:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

A higher F1 score indicates better performance, particularly when precision and recall are both high. These metrics are especially relevant when class distribution is skewed or when the cost of false positives and false negatives differs.

6.3.2. Confusion Matrix

The confusion matrix is a tabular summary of classification results. It displays the counts of true positive (TP), false positive (FP), true negative (TN), and false negative (FN) predictions. For a binary classification problem, the confusion matrix is a 2×2 matrix of the form:

$$\begin{bmatrix} \text{TP} & \text{FN} \\ \text{FP} & \text{TN} \end{bmatrix}$$

Each cell corresponds to a specific prediction outcome. From this matrix, additional metrics can be derived. For example, *specificity* measures the true negative rate:

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

Normalizing the matrix enables interpretation in terms of proportions, helping to identify potential biases in the model's predictions.

6.3.3. ROC Curve and AUC

Receiver Operating Characteristic (ROC)

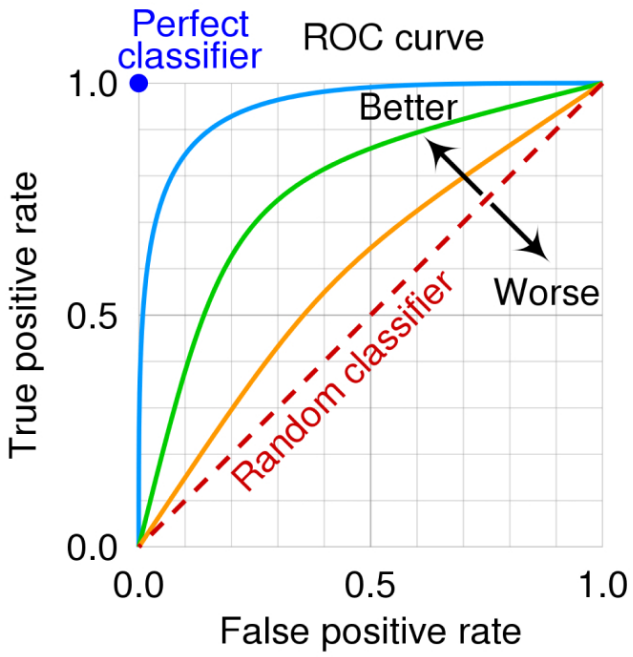

Figure 6.5: Visualization of the ROC curve. The x-axis represents the false positive rate (FPR), while the y-axis represents the true positive rate (TPR). The diagonal line represents a random classifier, whil the linies above it represent classifiers with varying levels of performance.

The Receiver Operating Characteristic (ROC) curve illustrates the diagnostic ability of a classifier as its decision threshold is varied. For each threshold—i.e., the probability

above which a sample is classified into a given class—the true positive rate (TPR) and false positive rate (FPR) are computed:

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad \text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

Each (FPR, TPR) pair is plotted as a point in 2D space, with FPR on the x-axis and TPR on the y-axis. The collection of these points forms the ROC curve. A perfect classifier corresponds to the point (0, 1), indicating no false positives and all true positives, while a random classifier produces a curve along the diagonal from (0, 0) to (1, 1). Example of such curves is shown in figure 6.5.

Area Under the Curve (AUC)

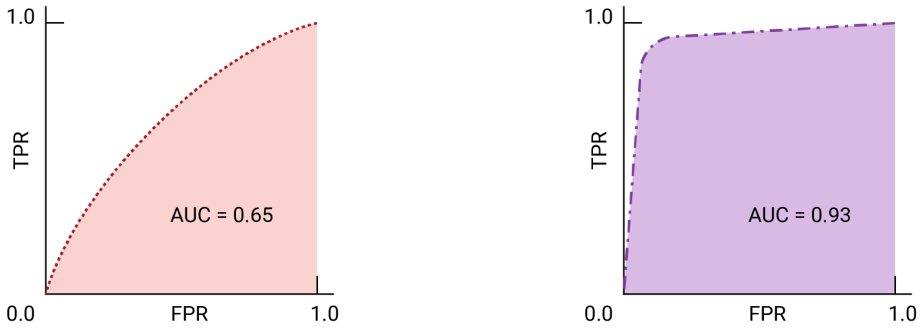

Figure 6.6: ROC and AUC of two hypothetical models. The curve on the right, with a greater AUC, represents the better of the two models.

The Area Under the Curve (AUC) summarizes the ROC curve in a single scalar value between 0 and 1. An AUC of 0.5 indicates random guessing, while an AUC of 1.0 represents perfect classification. AUC can be interpreted as the probability that the model, if given a randomly chosen positive and negative example, will rank the positive higher than the negative. The model with greater area under the curve is generally the better one.

Chapter 7

Convolutional Neural Networks (CNNs)

Convolutional neural networks (CNNs) are a class of machine learning algorithms designed to automatically and adaptively learn spatial hierarchies of features, ranging from low-level edges to high-level semantic patterns. They consist of stacked convolutional and pooling layers responsible for feature extraction, followed by fully connected layers that perform the final classification. Each layer passes its output to the next, enabling progressively more abstract and complex features to be learned as information flows through the network.

7.1. Convolutional Layer

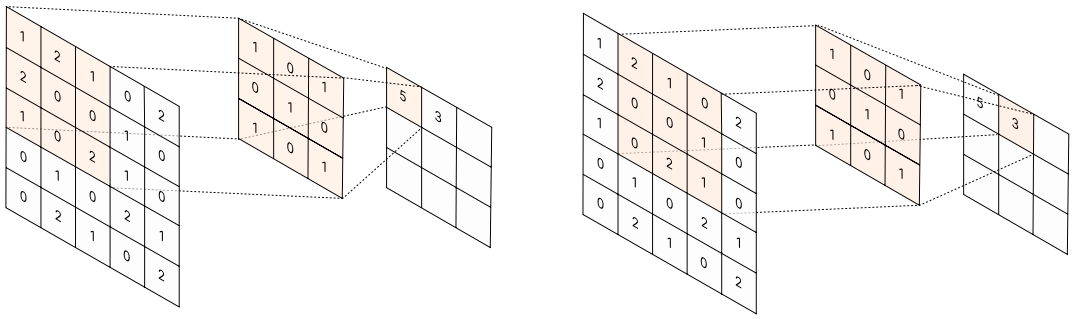

Figure 7.1: Example of convolutional layer operation

General Mechanism

A convolution is a linear operation that involves element-wise multiplication between an input array and a corresponding kernel (also referred to as a *filter*), followed by summation of the resulting products. The kernel is an array of values, typically smaller than the input array, that defines the operation and characterizes the specific features being extracted.

In a convolutional layer, multiple such kernels are applied to every valid subsection of the input array with dimensions matching the kernel (as shown in figure 7.1). The result of each convolution is a single scalar value, which becomes an element of the output feature map. To preserve or control the spatial dimensions of the output, *padding* may be applied to the input array, typically by adding zeros around its border.

The output of a convolutional layer is an array (or tensor) with reduced spatial dimensions (depending on the padding and stride) and a number of channels equal to the number of kernels employed in the layer.

Feature Recognition Mechanism

During backpropagation, each element of every kernel is treated as a learnable parameter and is optimized using gradient descent. The gradients of these parameters are computed with respect to each input element (more precisely, with respect to each considered subsection of the input array). These gradients are then aggregated—typically by averaging—and the kernel parameters are updated accordingly.

The general objective is for the kernels to adjust themselves to specific features of the input image—that is, to produce high activation values when a particular feature is present and low values when it is absent. The calculated gradients for each subsection of the input array depend on the values of the inputs. Consequently, it is assumed that during training, the elements of the kernels are updated in such a way that each kernel learns to recognize a distinct feature in the input array.

Figure 7.2 shows visualizations of features that maximally activate a selection filters across different convolutional layers of a CNN taken from [**?**].

In the early layers (e.g., `block1conv2`, `block2conv2`), the filters detect simple low-level features such as edges, colors, and textures. In deeper layers (e.g., `block4conv3`, `block5conv3`), the patterns become increasingly complex, capturing higher-level concepts.

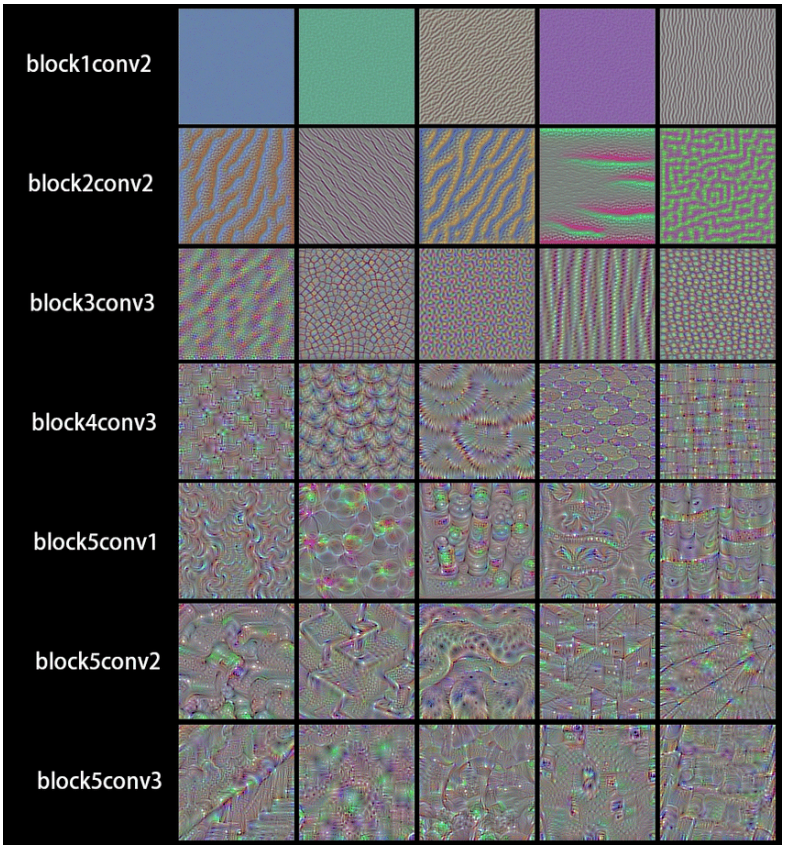

Figure 7.2: Visualization of feature maps learned by different convolutional layers of a CNN. Each row corresponds to a different convolutional layer (from early to late), and each column shows a visualization of the input pattern that maximally activates a specific filter. [**?**]

7.2. Pooling Layer

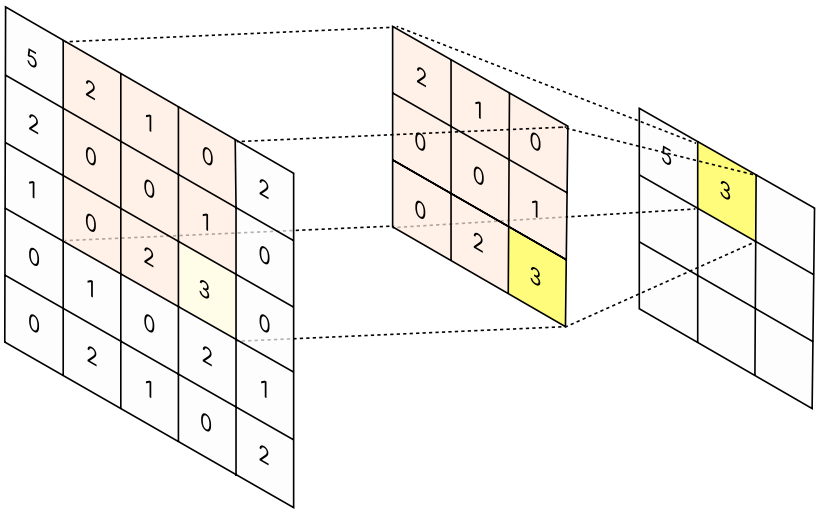

Figure 7.3: Caption for conv2.pdf

A pooling layer is a downsampling operation used to reduce the spatial dimensions of feature maps while preserving salient information. The specific pooling method used in each layer is selected based on its effectiveness for the given classification task, typically determined through empirical evaluation. Among the available techniques, *max pooling*—which selects the maximum value within a specified region—is the most commonly employed approach.

Pooling layers operate independently on each feature map (channel), where a channel corresponds to a set of activations produced by a single convolutional filter, and do not involve learnable parameters. By summarizing local regions, they can ensure invariance to small translations and reduce computational complexity in subsequent layers. Similar to convolutional layers, padding and stride can be used to control the spatial dimensions of the output.

7.3. Multi-Layer Perceptron

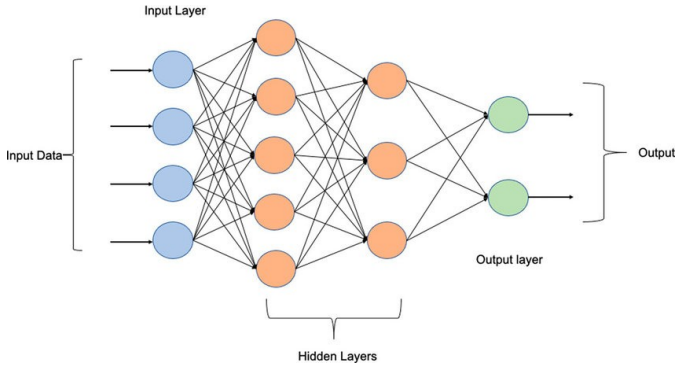

Figure 7.4: Illustration of a multi-layer perceptron architecture

A multi-layer perceptron (MLP) is a neural network composed of multiple fully connected layers (ie. a layer where each neuron receives input from every neuron in the previous layer). The MLP used in this context typically consists of several layers with progressively smaller numbers of neurons, culminating in an output layer tailored to the classification task.

In classical CNN architectures, the output of the final convolution or pooling layer is flattened and passed into a multi-layer perceptron to perform the final classification.

7.4. Complete CNN

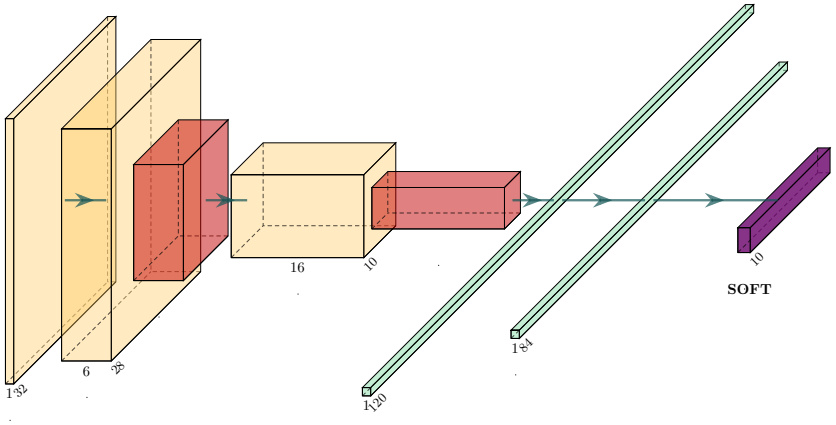

Figure 7.5: Example layout of a simple CNN

The first layers of the neural network consist of a hierarchy of convolution and pooling layers, each taking the output of the previous layer as input. The output of the final convolutional or pooling layer is then passed to a fully connected layer, which performs the final classification.

During training, the network first attempts to classify a batch of input elements. The predicted output is then compared to the true class labels, and a loss value is computed. This loss is subsequently backpropagated through the network, and the gradients for all learnable parameters are calculated. Finally, a step of gradient descent is performed, and the learnable parameters are updated accordingly.

7.5. Networks used

Two different architectures were used to classify the images, specifics of which are described below. Both networks were trained on the same dataset, with the same hyperparameters, to ensure a fair comparison of their respective performance.

7.5.1. LeNet-like network

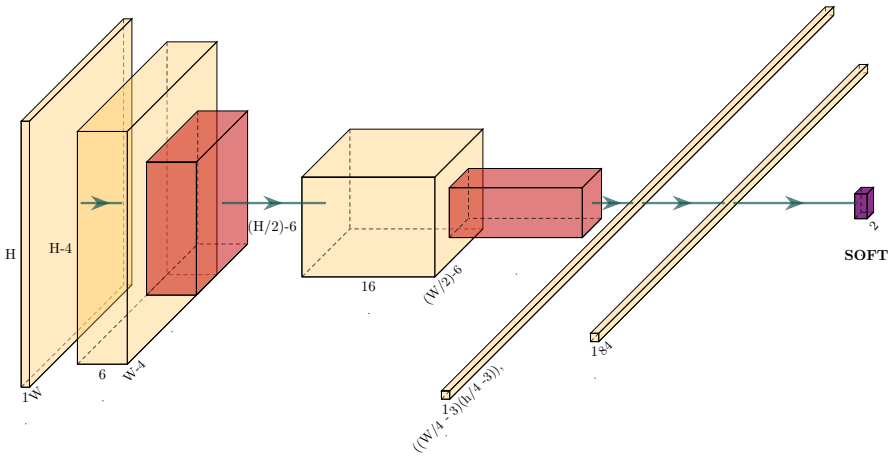

Figure 7.6: Graphical representation of the LeNet-inspired CNN used

Table 7.1: Summary of convolutional and pooling layers in the `SimpleCNN` architecture.

Layer Type	**Output Channels**	**Kernel Size**	**Stride**	**Output Size**
Conv2d (conv1)	6	5×5	1	28×28
AvgPool2d (pool1)	-	2×2	2	14×14
Conv2d (conv2)	16	5×5	1	10×10
AvgPool2d (pool2)	-	2×2	2	5×5
Conv2d (conv3)	120	5×5	1	1×1

One of the architectures used was based on the classical LeNet-5 convolutional neural network, originally developed for digit recognition. It is a shallow network originally designed for small-scale image classification tasks.

This architecture is summarized in Table 7.1 and visualized in figure 7.6.

7.5.2. AlexNet-like CNN

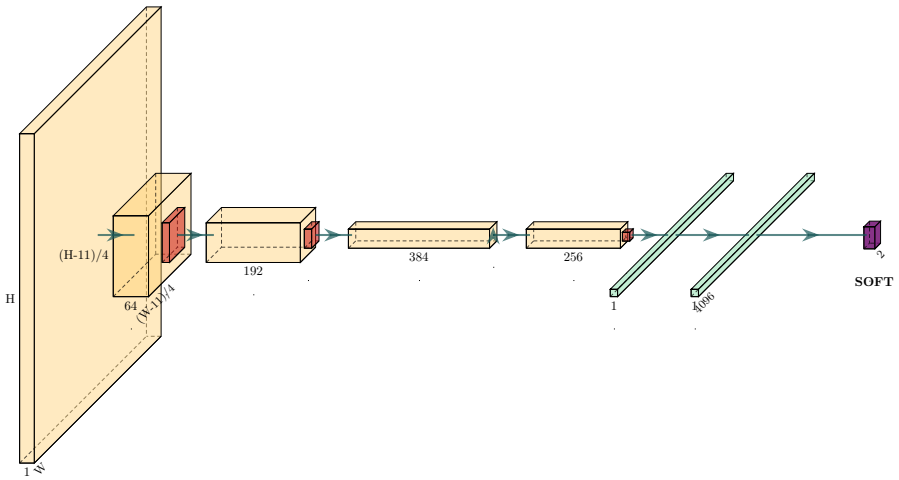

Figure 7.7: Example layout of a simple CNN

Table 7.2: Summary of convolutional and pooling layers in the `AlexCNNLayer` architecture.

Layer Type	Output Channels	Kernel Size	Stride	Padding
Conv2d (conv1)	64	11×11	4	2
MaxPool2d (pool1)	-	3×3	2	0
Conv2d (conv2)	192	5×5	1	2
MaxPool2d (pool2)	-	3×3	2	0
Conv2d (conv3)	384	3×3	1	1
Conv2d (conv4)	256	3×3	1	1
Conv2d (conv5)	256	3×3	1	1
MaxPool2d (pool3)	-	3×3	2	0

Another architecture utilized was a modified version of the AlexNet model, originally developed for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). The general hope being that the fact that it's significantly deeper and wider than LeNet would enable classify the images more effectively.

This architecture is summarized in Table **??** and visualized in figure 7.7.

7.5.3. Training Procedure

Both networks were trained using a training routine implemented in PyTorch. The training process operated on a combined dataset and dynamically inferred input dimensions from a sample image. Image preprocessing included optional downscaling using

nearest-neighbor interpolation, followed by conversion to tensor format. The models were optimized using stochastic gradient descent (SGD) with a fixed learning rate of 0.0005 and a momentum of 0.9. Cross-entropy loss was used as the objective function. Training was conducted for 10 epochs with a batch size of 16. During each epoch, loss and classification accuracy were recorded.

The dataset was split into training, validation, and test sets in a 70/15/15 ratio. Networks were trained on the training set, and their performance was evaluated on the validation set. The best-performing model on the validation set was selected for final evaluation on the test set, which consisted of data the model had no access to previously. Training was conducted separately on the full dataset and on a subset limited to the top 50% of most energetic lepton interactions. For simplicity, only data from plane2 was used. Input images were scaled down to 0.33 and 0.5 of their original size. Confusion matrices were computed for all trained models, and results were compared. F1 scores and ROC curves were calculated for the best-performing models. Results of the training are described in the next chapter.

7.6. Results

7.6.1. LeNet

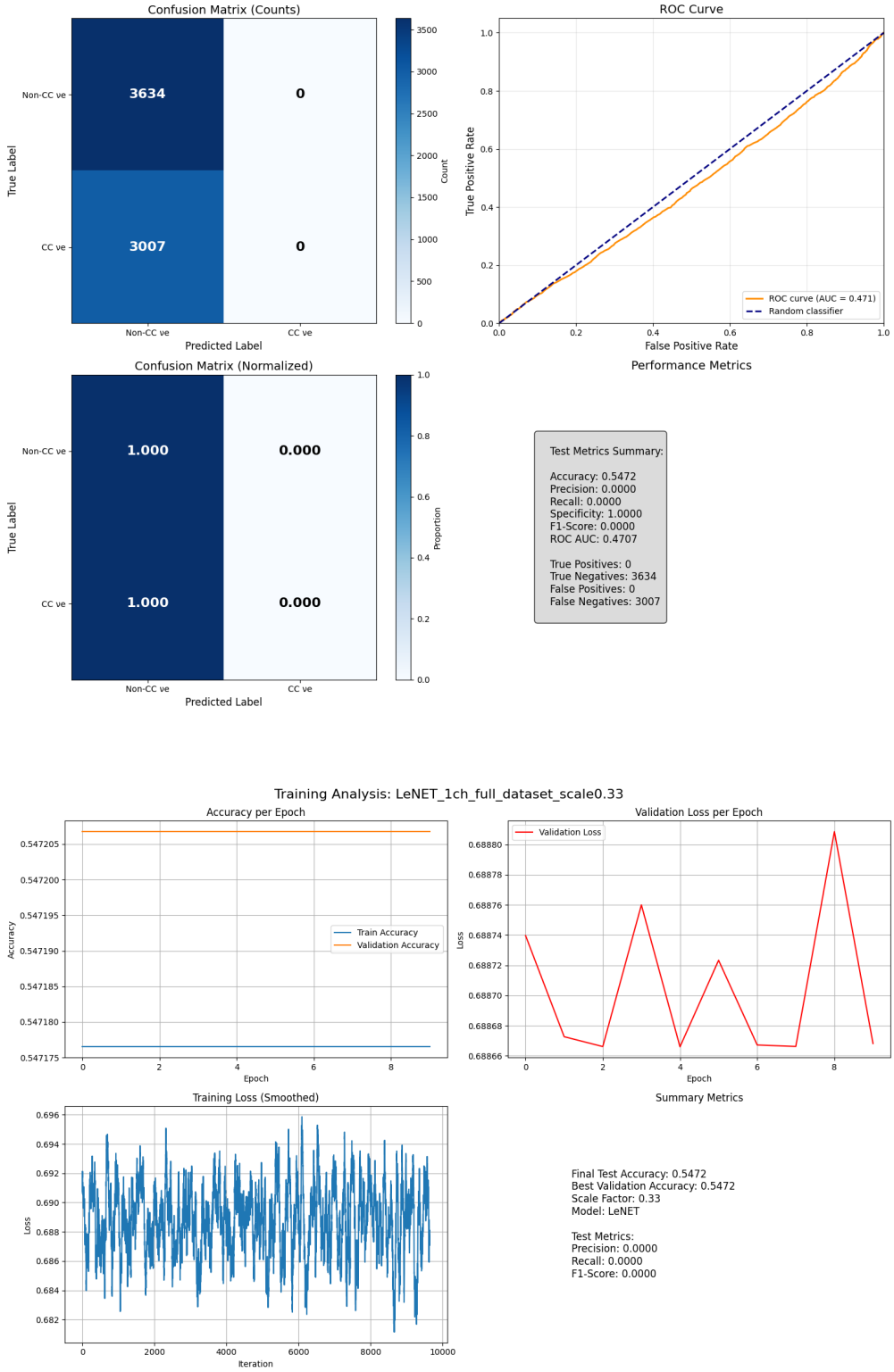

Figure 7.8: LeNET training results on the full dataset with images scaled down to 0.33 of their original size.

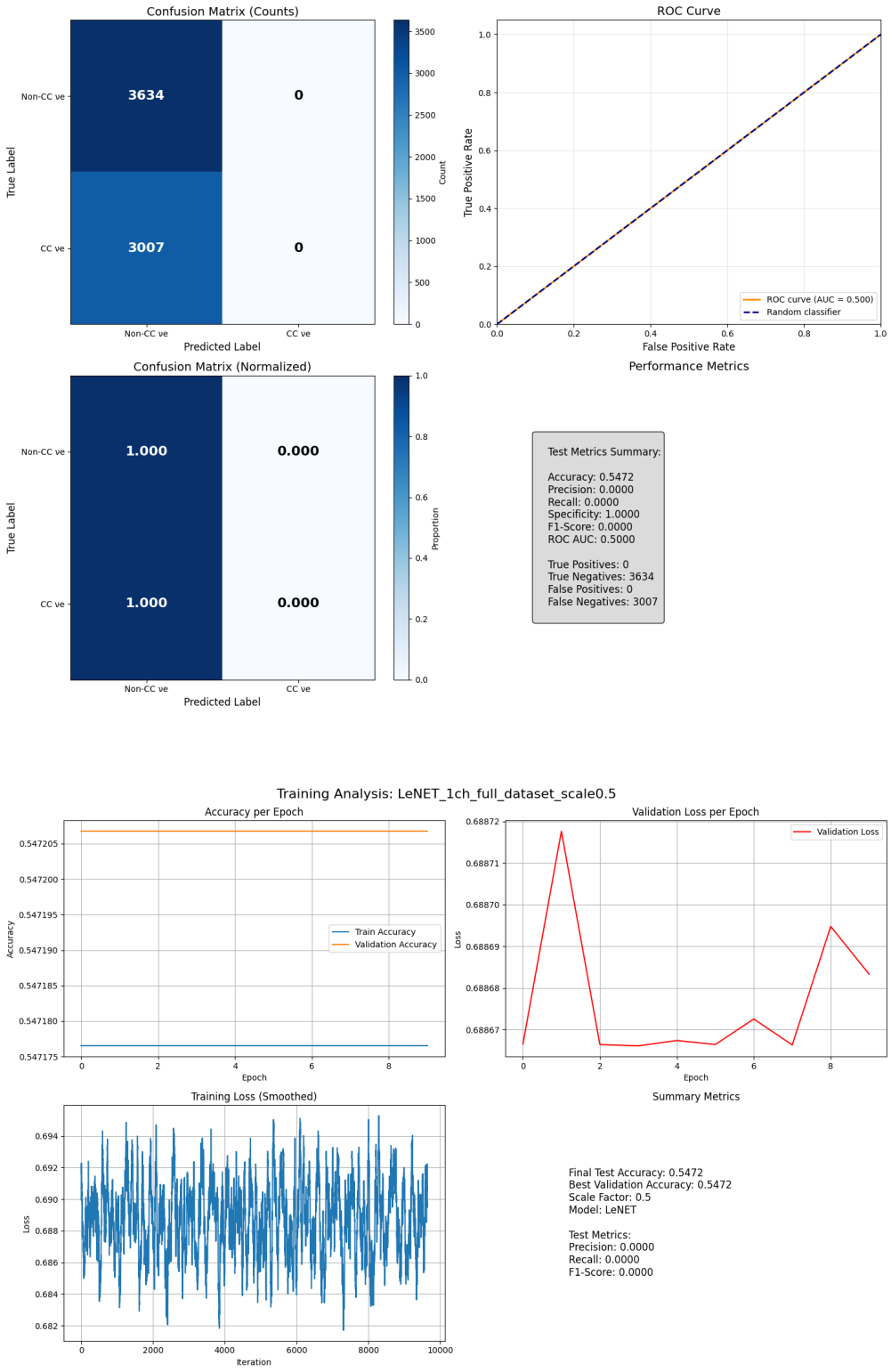

Figure 7.9: LeNET training results on the full dataset with images scaled down to 0.5 of their original size.

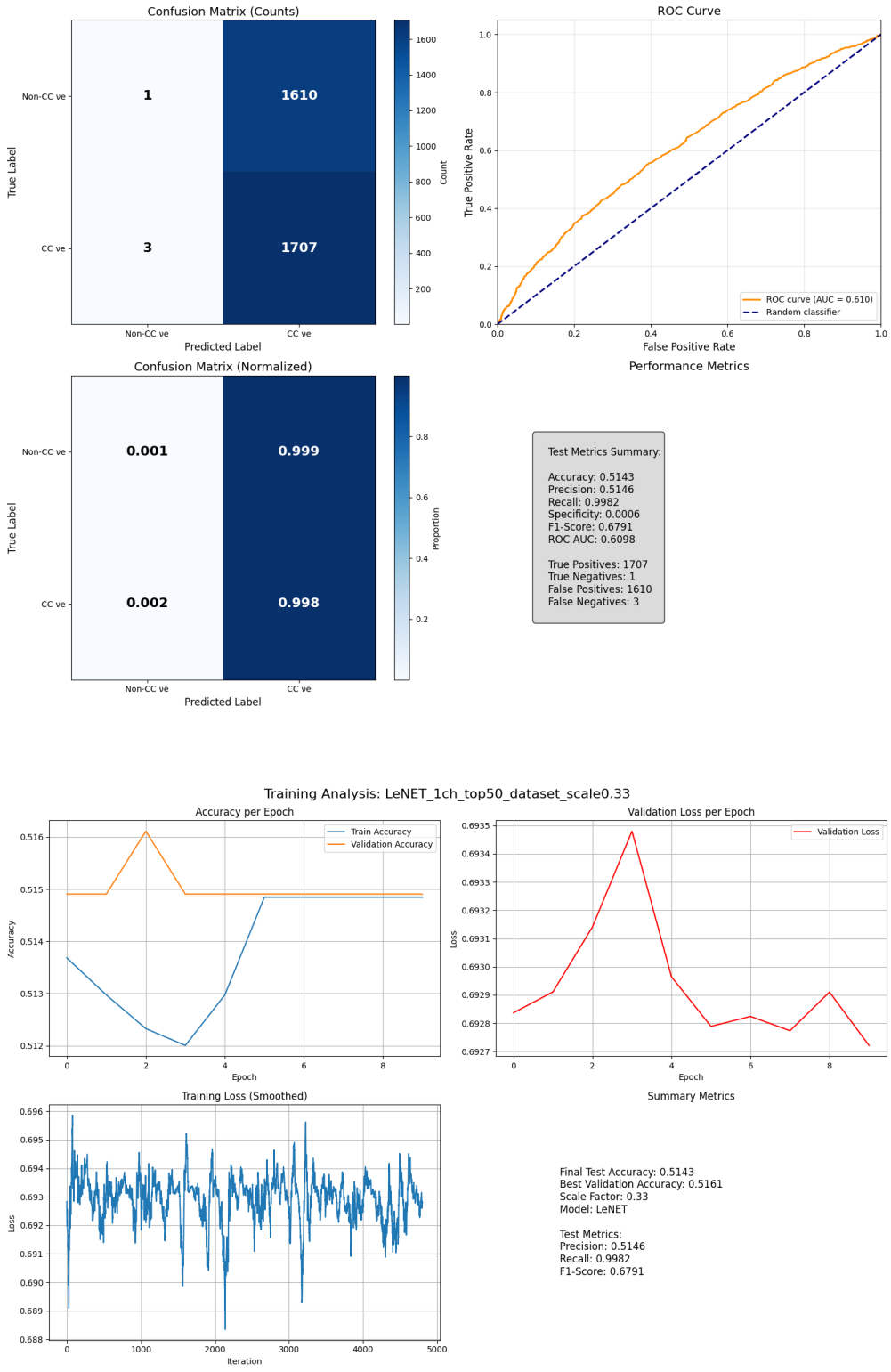

Figure 7.10: LeNET training results on the top50 dataset with images scaled down to 0.33 of their original size.

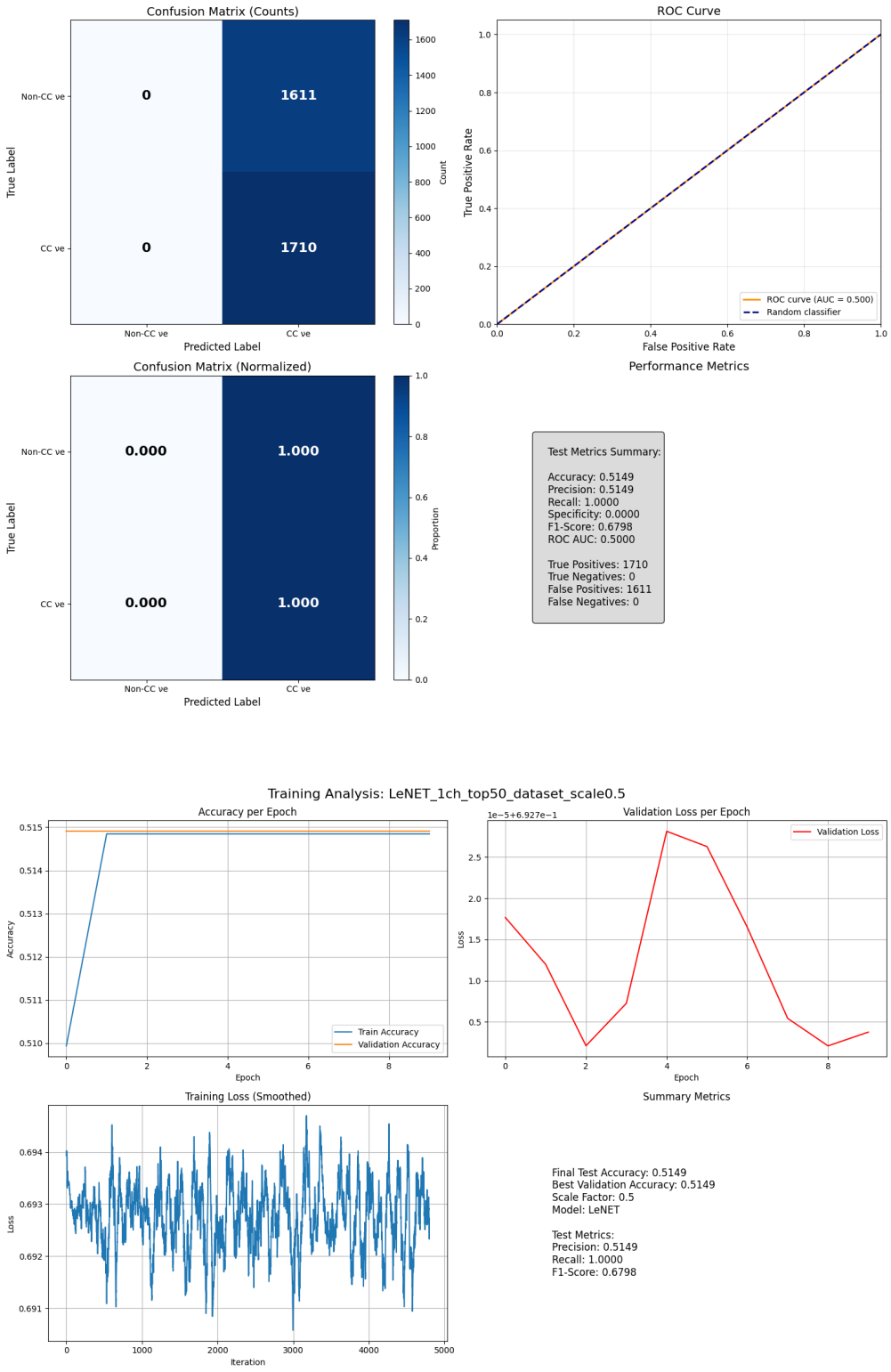

Figure 7.11: LeNET training results on the top50 dataset with images scaled down to 0.5 of their original size.

Training results for the LeNet-like network are presented in Figures 7.8, 7.9, 7.10, and 7.11. The network was unable to learn any meaningful features from the data, as evidenced by the fact that it classified all events as neutrino interactions. The training loss did not decrease and simply oscillated around a constant value. The classification accuracy remained close to 0.5 for all runs. It can be reasonably concluded that the network did not learn any non-trivial features from the data, and training was completely unsuccessful.

7.6.2. AlexNet

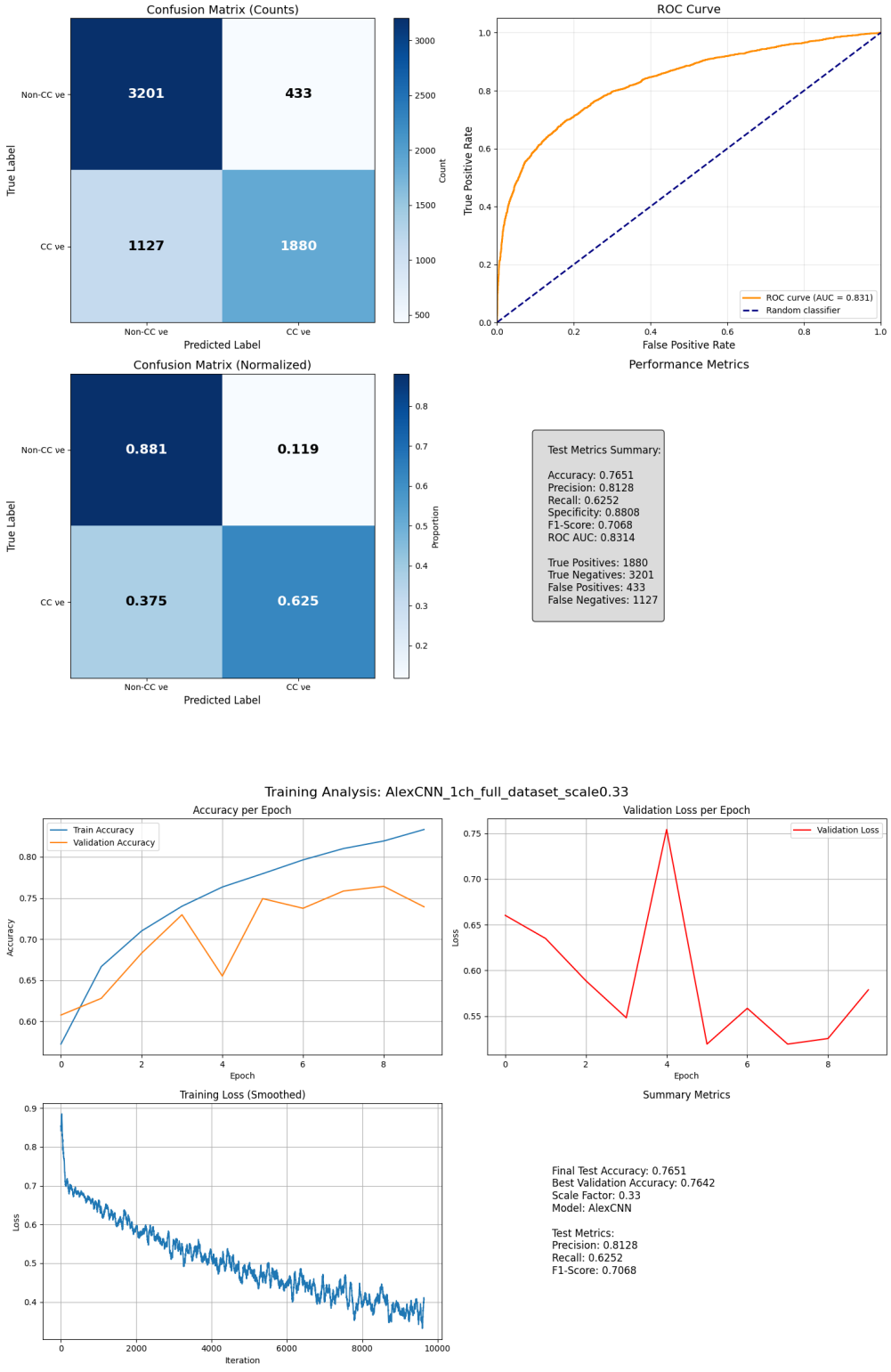

Figure 7.12: Alexnet training results on the full dataset with images scaled down to 0.33 of their original size.

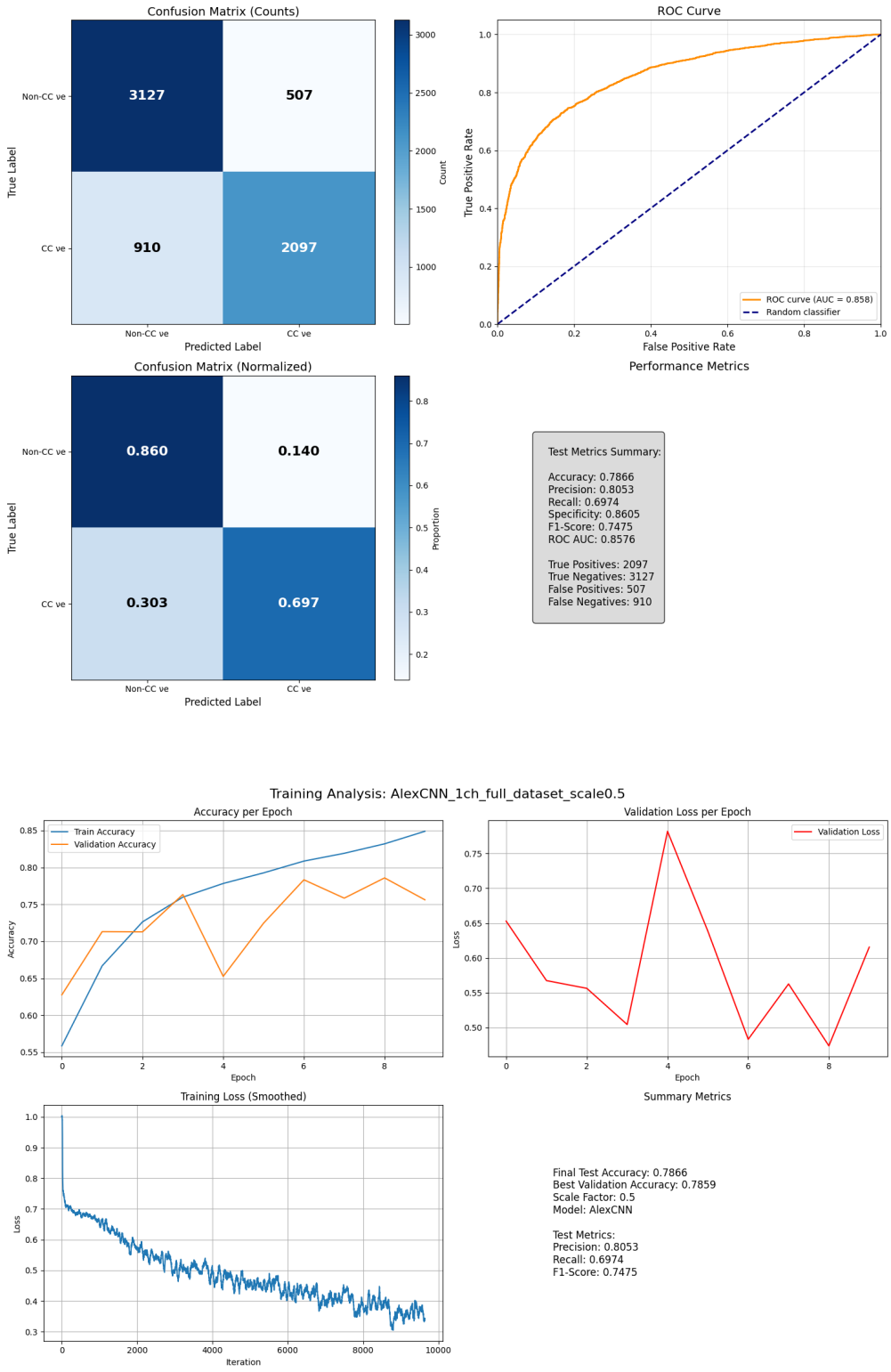

Figure 7.13: Alexnet training results on the full dataset with images scaled down to 0.5 of their original size.

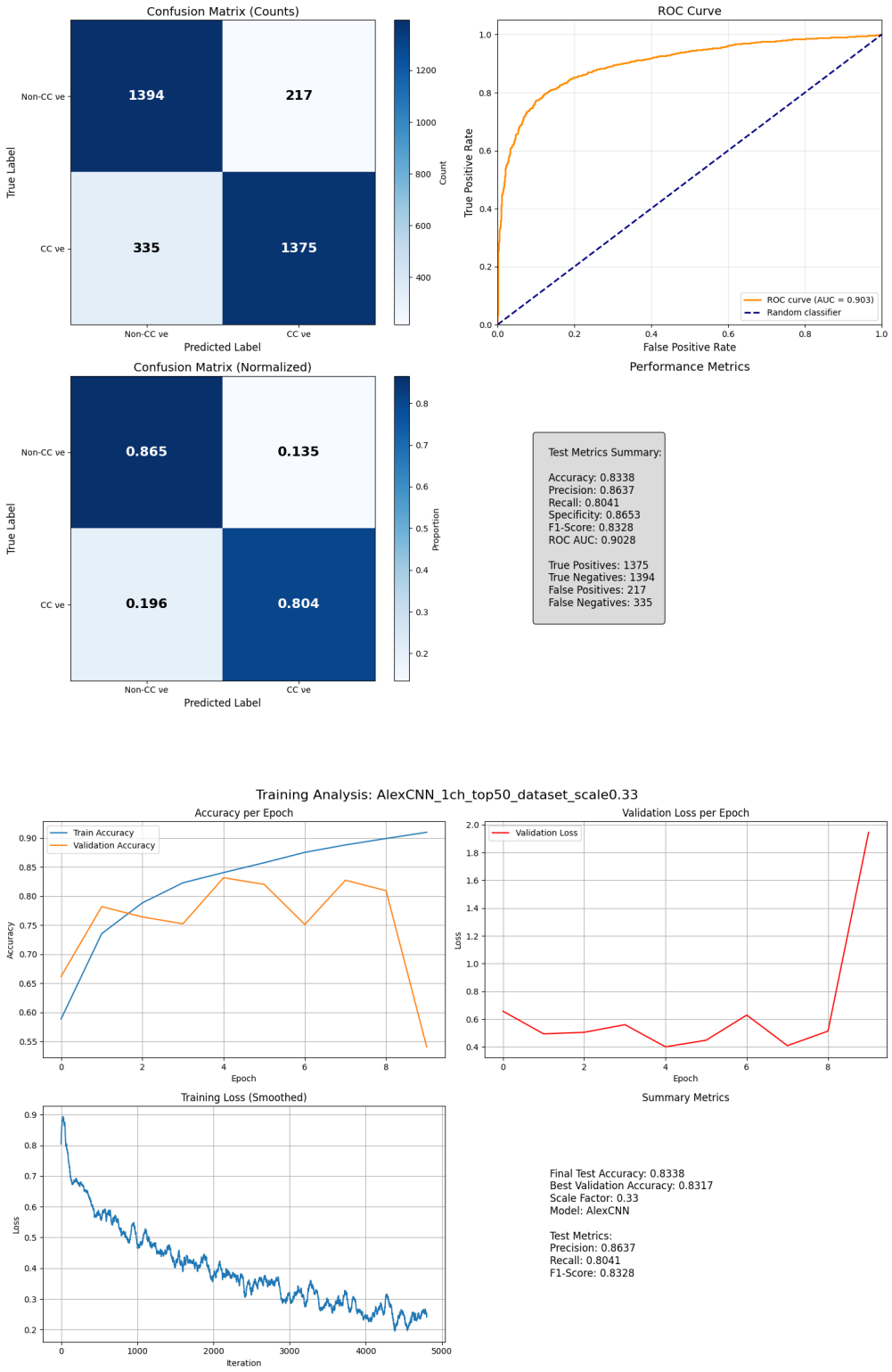

Figure 7.14: Alexnet training results on the top50 dataset with images scaled down to 0.33 of their original size.

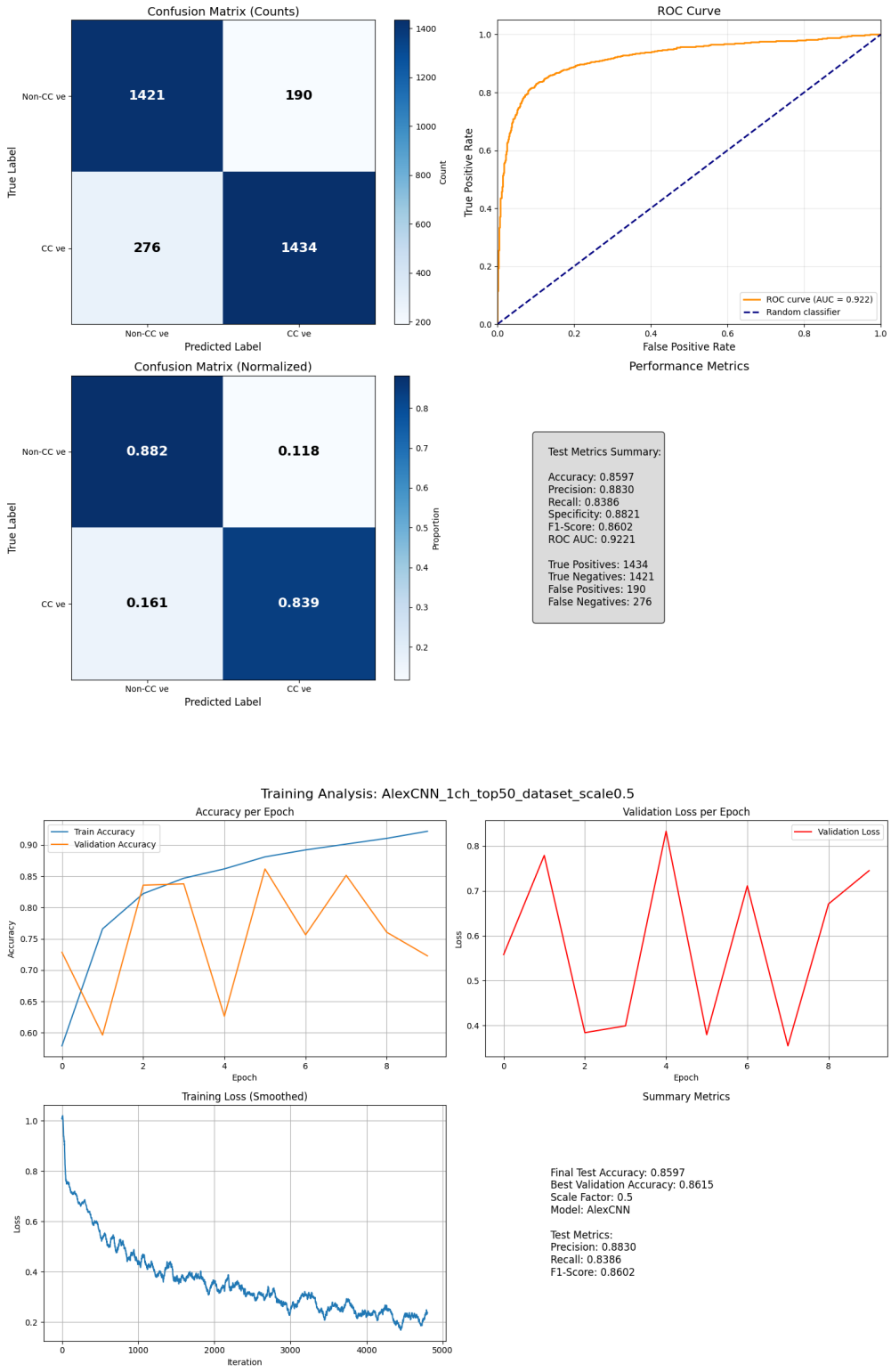

Figure 7.15: Alexnet training results on the top50 dataset with images scaled down to 0.5 of their original size.

Training results for the AlexNet-like network are presented in Figures 7.12, **??**, 7.14, and 7.15. Overall, the network was able to achieve satisfactory results for all datasets and scaling factors. The performance metrics achieved for each training suggest that some non-trivial underlying patterns were learned by the network. Training loss behaved as expected—decreasing steadily, with the rate of change becoming smaller with each epoch. Classification accuracy increased over the course of training.

In all cases, AlexNet-based networks performed better at classifying non-CC events than CC events (the number of false positives in the latter case was greater in every training). Increasing the scale factor of the input image did increase the network's performance, but not in a very significant way. Most of the improvement came from reducing the number of false negatives for electron neutrino interactions.

7.6.3. discussion

As noted before, the LeNet-like network was unable to learn any meaningful features from the data, while the AlexNet-like network performed relatively well on all datasets. This outcome could have been somewhat expected, as LeNet is significantly less complex and shallower than AlexNet, which limits its ability to learn complex features from the data. It may be possible that the learning results could be improved by adjusting the training parameters, but the complete lack of performance suggests that it is not well-suited for the classification task at hand.

The strong performance of the AlexNet-like network suggests that it was able to learn some non-trivial features from the data. The fact that the network ended up being slightly better at classifying non-CC events than CC events can also be considered expected behavior—certain noise patterns (such as the presence of tracks from other leptons or overlapping noise tracks) may be misinterpreted as electron showers. The performance of the network could still be improved by refining the training parameters, but the observed rise in validation loss during later training epochs suggests that the network may have started to overfit the training data. Training on a larger dataset could potentially bring a meaningful performance increase.

Taking into account the high accuracy of the AlexNet-like network, it can be reasonably assumed that the classification problems, as previously stated, are solvable using deep learning techniques. To extend the discussion, another approach to image classification was attempted—namely, one involving an image transformer architecture. The details of this approach are described in the next chapter.

Chapter 8

Transformer

The transformer is a deep learning architecture introduced for processing sequential data, relying entirely on attention mechanisms and eliminating recurrence. It enables high parallelization and long-range dependency modeling, making it suitable for a wide range of tasks, including computer vision.

The core components of the transformer include multi-head self-attention layer and residual connections with layer normalization. The model is organized into repeated layers or blocks, each consisting of an attention sublayer (see Figure 8.2).

Inputs to the transformer are first passed through an embedding layer, and positional encodings are added to the embeddings to retain information about the order of tokens in the sequence. These embeddings are then passed through a number of encoder blocks, each consisting of multi-head self-attention and linear layers, with the output of one block serving as the input to the next. The final output is a sequence of continuous vectors, which can then be processed by a decoder or used directly for tasks such as classification or regression.

8.0.1. Embeddings

Transformer-based architectures operate on continuous vector representations. For this reason, each input passed into the network—also known as a token—must be mapped to a fixed-dimensional dense vector, known as an embedding. An example of word embeddings in a low-dimensional space is shown in Figure8.1.

Formally, for an input consisting of V tokens and an embedding dimension d, the embedding layer is represented by a matrix $\mathbf{E} \in \mathbb{R}^{V \times d}$.

Given a sequence of input tokens $\{t_1, t_2, \ldots, t_n\}$, the corresponding embedding vectors $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ are obtained by indexing into the embedding matrix:

$$\mathbf{x}_i = \mathbf{E}[t_i] \in \mathbb{R}^d$$

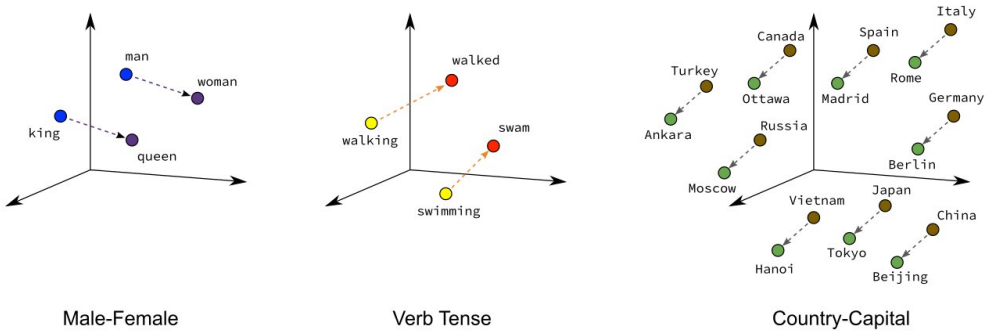

Figure 8.1: A simplified example of word embeddings projected into three dimensions. Words that share functional or structural roles—such as verb tense or country-capital pairs—tend to form consistent spatial patterns.

The embedding matrix consists of trainable parameters and is optimized jointly with the rest of the model. The goal of this mapping is to project discrete inputs into a continuous space where meaningful relationships can be learned during training.

8.0.2. Transformer Encoder

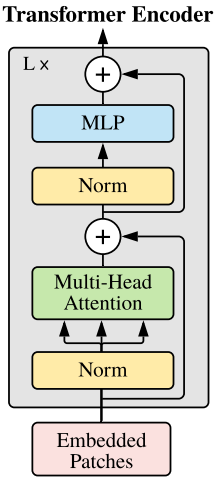

Figure 8.2: Visual representation of a transformer block.

This structure is referred to as an encoder block and is visualized in Figure 8.2. In the transformer architecture, the self-attention mechanism—specifically multi-head attention—is a central component that enables the model to attend to different parts of the input sequence simultaneously.

8.0.3. Single attention head

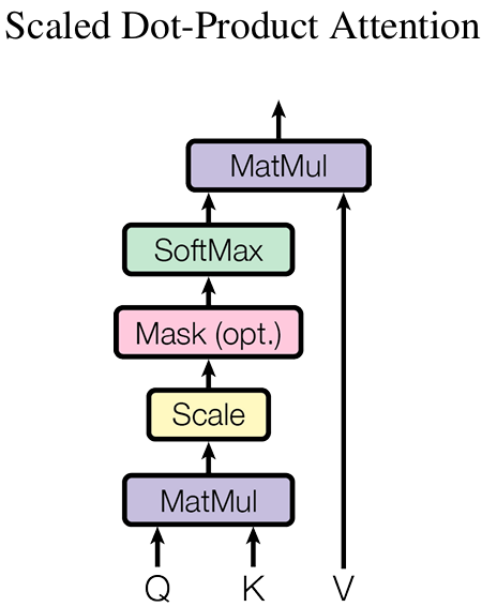

Figure 8.3: Graphic representation of a single attention head. The input query, key, and value matrices are passed through a scaled dot-product attention mechanism, which computes attention scores and produces the output.

The mechanism of a single attention head is illustrated in Figure 8.3.

Each attention head receives three inputs: the query vector $Q \in \mathbb{R}^{n \times d_k}$, the key matrix $K \in \mathbb{R}^{n \times d_k}$, and the value matrix $V \in \mathbb{R}^{n \times d_v}$, where n is the sequence length, and d_k, d_v are the dimensionalities of the key and value vectors, respectively. The attention operation begins by computing a score vector through the dot product of the query and the transpose of the key matrix:

$$\text{scores} = QK^\top$$

This score vector is then scaled and passed through a softmax function to yield attention weights:

$$\text{weights} = \text{softmax}\left(\frac{\text{scores}}{\sqrt{d_k}}\right)$$

These weights are subsequently used to compute a weighted sum of the value vectors, producing the output of the attention head:

$$\text{Attention}(Q, K, V) = \text{weights} \cdot V$$

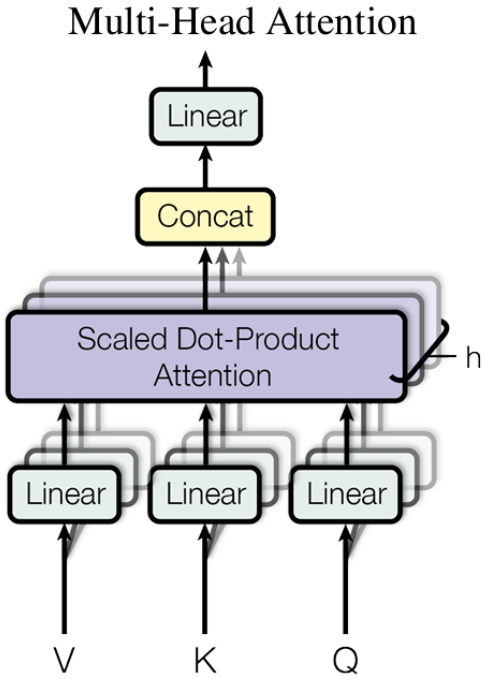

Figure 8.4: Graphic representation of a multi-head attention mechanism. The input query, key, and value matrices are passed through multiple independent attention heads, each computing its own attention output. The outputs of all heads are concatenated and projected back to the original dimensionality.

8.0.4. Multi-head attention

A visual overview of the multi-head attention mechanism is provided in Figure 8.4.

The multi-head attention mechanism extends the single-head idea by introducing multiple independent attention heads. Before attention is computed, the original input matrices Q, K, and V are each passed through separate trainable linear projections.

Instead of applying a single attention function with d_{model}-dimensional queries, keys, and values, the multi-head attention mechanism projects the inputs into multiple lower-dimensional subspaces and performs attention in parallel across these subspaces.

Specifically, for each attention head $i \in \{1, \ldots, h\}$, the queries Q, keys K, and values V are linearly projected using trainable weight matrices:

$$QW_i^Q \in \mathbb{R}^{n \times d_k}, \quad KW_i^K \in \mathbb{R}^{n \times d_k}, \quad VW_i^V \in \mathbb{R}^{n \times d_v}$$

Each attention head computes scaled dot-product attention independently:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

The outputs of all attention heads are then concatenated along the feature dimension:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

where $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.

The matrices W_i^Q, W_i^K, W_i^V, and W^O consist of trainable parameters optimized during training.s

8.0.5. Residual connections and layer normalization

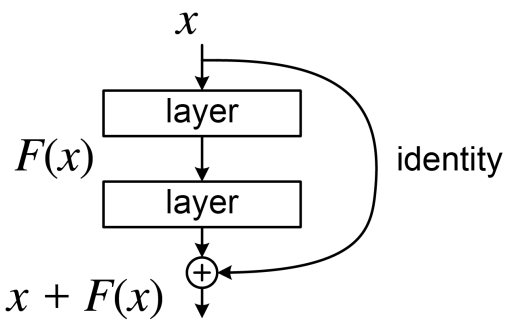

Figure 8.5: Graphic representation of a residual connection.

The residual connection and normalization process are shown in Figure 8.5.

To facilitate stable training and improve gradient flow, the transformer applies residual connections around each sublayer, followed by layer normalization.

Given an input $\mathbf{x}$ and a sublayer function $\text{Sublayer}(\cdot)$, the output is computed as:

$$\text{LayerNorm}(\mathbf{x} + \text{Sublayer}(\mathbf{x}))$$

The residual connection adds the input of the sublayer to its output, and layer normalization standardizes the summed output across the feature dimension. This structure helps preserve information and stabilizes the optimization process.

8.0.6. Positional encoding

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$
$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

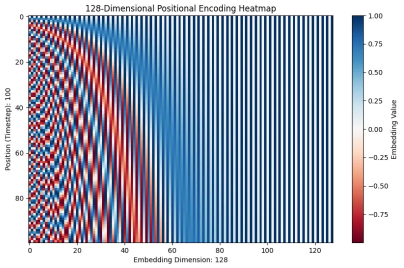

Figure 8.6: Example of how positional encoding can be handled. The heatmap on the right visually represents a positional encoding for 128-dimensional vectors. Each row corresponds to a position in the input sequence, and each column represents a dimension within the embedding vector. The color gradient signifies the value of the positional encoding at that specific position and dimension. The equations on the left define the sinusoidal functions used to compute the positional encoding for each position pos and dimension i in the embedding vector.

The sinusoidal positional encoding and its heatmap representation are shown in the diagram in Figure **??**.

Importantly, the architecture described above does not inherently retain positional information of the input sequence. To address this, the transformer architecture incorporates *positional encoding*, which injects information about the position of each token in the sequence into the input embeddings. The encoded position of each token is added to its corresponding embedding before being passed through the transformer layers:

$$\mathbf{z}_0 = \mathbf{x} + \mathbf{P}$$

where $\mathbf{x}$ denotes the input embedding and $\mathbf{P}$ is the positional encoding.

8.1. Image Transformer

8.1.1. Image Transformer Architecture

The transformer architecture adapted for images is illustrated in Figure 8.7.

The image transformer extends the standard transformer architecture to process visual inputs. Rather than operating on word tokens, the input image is divided into a sequence of fixed-size non-overlapping patches. A typical configuration involves splitting the image into 16 patches of 16×16 pixels each. These patches are then flattened into vectors and linearly projected to a fixed embedding dimension, producing a sequence of patch embeddings.

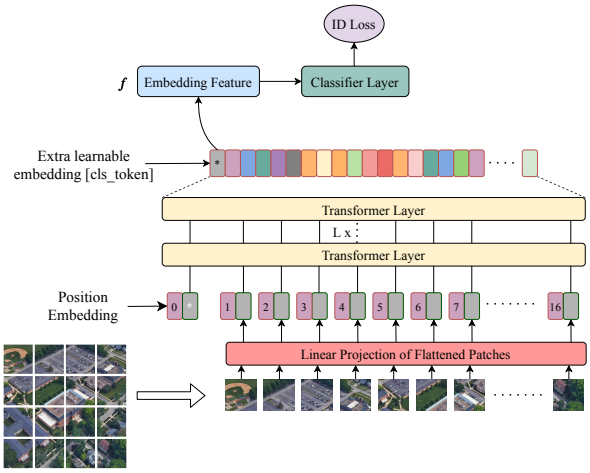

Figure 8.7: Visual transformer architecture visualized

Each patch embedding is augmented with a corresponding positional embedding to encode spatial location. This sequence is then processed by a stack of transformer encoder layers, each comprising multi-head self-attention and feed-forward sublayers with residual connections and layer normalization. The transformer outputs a set of encoded vectors, which are subsequently used for downstream tasks such as classification or detection.

8.1.2. Classification Token

To enable image-level classification, a special token known as the classification token, or [CLS] token, is prepended to the sequence of patch embeddings. This token does not represent any specific region of the image and is initialized as a trainable vector. Throughout the transformer layers, it interacts with other tokens via the self-attention mechanism and gradually aggregates global information about the entire image.

After processing through all transformer layers, the final output corresponding to the [CLS] token serves as a global representation of the image. This output is passed to a multi-layer perceptron (MLP) head to produce class logits. The [CLS] token is trained jointly with the model to optimize performance on the target classification task.

8.1.3. Positional Encoding for Image Fragments

Figure 8.8 shows how positional encodings correlate spatially in image transformers.

To retain spatial context after converting image patches into a sequence, positional embeddings are added to each patch embedding. These embeddings are learnable parameters and are optimized during training alongside the rest of the model. Their purpose is to encode the location of each patch within the original image, enabling the model to

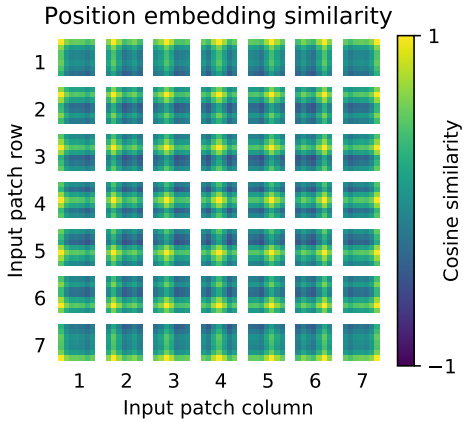

Figure 8.8: Cosine similarity of positional encoding for image patches from the original Vision Transformer paper. Embeddings for patches close to each other are similar.

distinguish between otherwise identical patches in different positions.

Empirical analysis has shown that these embeddings evolve such that neighboring patches in the image space tend to receive similar positional encodings. This spatial correlation helps the transformer capture local structure while also enabling long-range dependencies to be modeled across distant regions of the image.

8.2. Transformer architecture used

The specific transforemer image transformer that ended up being used divided images into patches of 16x16 pixels each. Each patch is mapped to a 32-dimensional embedding vector. Positional encoding is applied using sine and cosine functions across the embedding dimensions. The model includes a single multi-head self-attention layer with two heads to capture relationships between patches. A final three-layer multilayer perceptron consisted of layer sizes of 32, 16, and 2 neurons respectively.

8.2.1. Training procedure

The transformer was trained for up to 10 epochs using a dataset split into training and validation batches. The validation set consisted of 1000 batches taken from the dataset, while the rest was used for training. An Adam optimizer with an exponentially decaying learning rate starting at 0.001 was used, with the learning rate decreasing by 5% every 10 full passes through the training data. The model optimized a binary cross-entropy loss and tracked accuracy as a metric. Training incorporated early stopping, halting if the validation accuracy did not improve for 10 consecutive epochs.

8.3. Transformer training results and evaluation

Figure 8.9: Caption 1

Figure 8.10: Caption 2

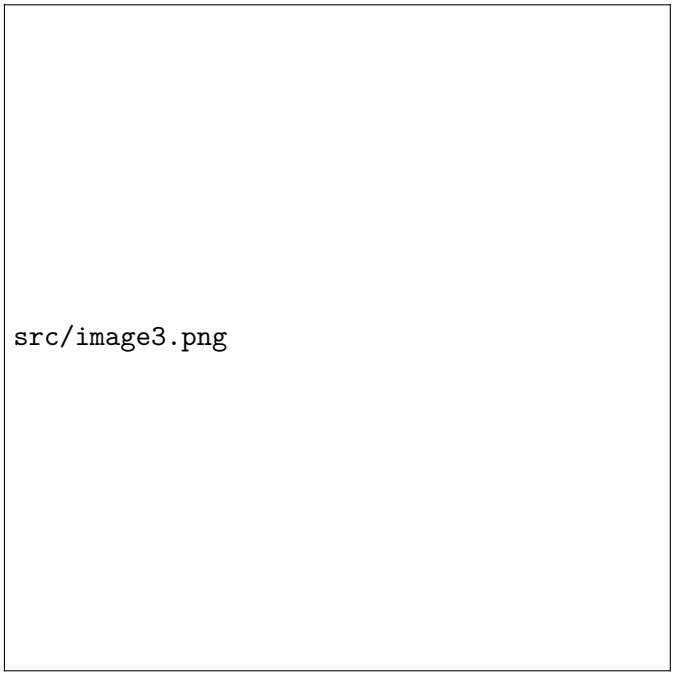

Figure 8.11: Caption 3

8.3.1. Training Summary

The neural network was trained for 10 epochs using an exponentially decaying learning rate schedule. The final training loss reached **0.5839**, with a training accuracy of **69.04%**. Validation performance at the end of training showed a loss of **0.6139** and an accuracy of **66.00%**, which was also the best recorded validation accuracy.

Testing on the held-out dataset yielded a final loss of **0.6218** and an accuracy of **66.03%**, indicating consistent generalization performance.

8.3.2. Learning Dynamics

During training, the loss decreased by **12.0%**, and accuracy improved by **9.75 percentage points**. The training-validation accuracy gap was **3.04 percentage points**, suggesting low overfitting. Loss fluctuations over the final five epochs were minimal (standard deviation: **0.0039**), indicating moderate convergence.

8.3.3. Classification Metrics

Evaluation of class-wise performance yielded the following:

- **Class "CC e"**
  - Precision: 0.6773
  - Recall: 0.6442
  - F1-Score: 0.6603
- **Class "other"**
  - Precision: 0.6441
  - Recall: 0.6773
  - F1-Score: 0.6603

The overall model achieved an **AUC-ROC score of 0.7207**, confirming its capacity to discriminate between the two classes effectively.

8.3.4. Confusion Matrix

Figure 8.12 presents the normalized confusion matrix for the test set. The model correctly classified **64%** of "CC e" instances and **68%** of "other" instances. Misclassification rates were relatively balanced between the two classes.

Figure 8.12: Normalized confusion matrix showing prediction distribution across classes.

8.4. Conclusion

8.5. Bibliography

.1. HDF5 File Structure

The complete structure of the HDF5 tables used in the MicroBooNE OpenSamples dataset is detailed below:

Table Name	Description
`/event_table`	Stores metadata for individual detector readouts and simulated neutrino interactions. Includes event identifiers, neutrino properties (energy, direction), interaction vertex, and interaction type (charged- or neutral-current).
`/hit_table`	Contains reconstructed hits from Gaussian pulses on wire waveforms. Includes hit positions, times, plane/wire IDs, and signal properties like integral and RMS.
`/edep_table`	Holds simulated true energy deposition data within the TPC. Entries link to the associated hit and originating simulated particle.
`/particle_table`	Stores detailed simulation data for particles from neutrino interactions as simulated by Geant4, including start/end positions, momenta, parent-child relationships, and process labels.
`/opflash_table`	Contains reconstructed optical flash data detected by PMTs, such as time, width, and spatial barycenter.
`/opflashsumpe_table`	Provides total photoelectron (PE) counts per PMT for each optical flash, useful for calibrated optical signal reconstruction.
`/ophit_table`	Details individual PMT hits, including amplitude, width, time, and associated PMT channel. Links to flash and PE sum tables.
`/pandoraHit_table`	Describes how individual TPC hits are grouped into clusters by the Pandora reconstruction framework. Includes associations to PFParticles and time slices.
`/pandoraPfp_table`	Stores data for Pandora-reconstructed PFParticles, which represent 3D hit clusters. Contains particle type IDs, vertex info, and classification scores.
`/pandoraPrimary_table`	Contains metadata for interactions (primaries) identified by Pandora, including neutrino/cosmic classification scores and optical-TPC flash matching quality.
`/wire_table`	Records full waveform data for each detector wire over 6400 time ticks. Provides raw ADC values for hit reconstruction. Only in "With Wire" datasets.

Table 1: Summary of tables and their contents in the detector data model.