UNIVERSITY
OF OSLO

Master's thesis

Deployment of unsupervised learning in the search for new physics at the LHC with the ATLAS detector

Search for heavy neutrinos at the LHC with the ATLAS detector

Sakarias Garcia de Presno Frette

Computational Science: Physics
60 ECTS study points

Department of Physics
Faculty of Mathematics and Natural Sciences

12th November 2022

Sakarias Garcia de Presno Frette

Deployment of unsupervised learning in the search for new physics at the LHC with the ATLAS detector

Search for heavy neutrinos at the LHC with the ATLAS detector

Supervisors:
Professor Farid Ould-Saada
Dr. Eirik Gramstad
Dr. James Catmore

Abstract

Acknowledgments

- Veilledere
- William og Mikkel
- Foreldre
- NN

Contents

Introduction

Outline of the Thesis

The master thesis is outlined in the following way. The first two chapters are dedicated to the neccesary machine learning and standard model physics background required to understand the analysis done and tools used in the thesis. The third chapter goes through the implementation of the project, where the datasets comes from, the ATLAS architecture, the programming libraries, feature choice, and so on. Chapter four goes through the results from the implementation. Chapter five is dedicated to the discussion and interpretation of the results, the pros and cons of teh implementation, aspects for future improvement, and other thoughts around the process. The final chapter is dedicated to the conclusion, were the foundings are summarized.

Chapter 1

Machine learning

Anomaly detection

Anomaly detection is a tool with a wide range of uses, from time series data, fraud detection or anomalous sensor data. Its main purpose is to detect data which does not conform to some predetermined standard for normal behavior. The predetermined standard varies from situation to situation, both from the context it self and what is expected as an anomaly. Anomalies are typically classified in three categories [1]:

1. Point anomalies
2. Contextual anomalies
3. Collective anomalies

Point anomalies are singular or few outliers from a larger contect or group. These anomalies can occur in many situations, are indeed quite important to detect. One such example is Michael Phelps. Phelps is famous for being one of the best swimmers of all time. Along with extensive training, planning and dedication, he has another tool that has helped him, he does not produce much lactic acid. In fact, his body produces so little that he can swim continously and much more intensive than most other top swimmers. This ability is not common, infact it is very rare amongst humans, and can be considered a point anomaly. It is important to understand that point anomalies does not have to be singular occurances. Rather they are extremly rare events that deviate alot from the expected behavior.

Contextual anomalies are another kind of anomalies, and are defined based on the context of the anomaly and data, rather than as a whole. Suppose you have have data on continous stream of gas in a pipe. The extraction of this gas is day dependent, to the point where the delivery on saturdays might oscillate between half and 3/4 of that of monday through friday, due to shorter work day. Should there one saturday suddenly flow the same amount as friday, an analyst think nothing of this, but due to this being a saturday, the context of this behavior dictates that this be categorized as a contextual anomaly.

The last type of anomaly described by Chandola, Banerjee and Kumar [1] are the collective anomalies. The collective anomalies are anomalies that as a group deviate from the expected behavior of the dataset. In particle physics these anomalies are the only type that are of interest. This is because there are so many sources for anomalous behavior in an experiment that only collective one are worth investigating. The major problem for such experiments is noise, and noise can be created from a large number of components. This alone is reason enough to only consider collective anomalies. Another reason is that certain processes in particle physics look much alike, but have different crossection, thus one process is much more likely to happen than another. This was one of the main issues with the discovery of Higgs, as Higgs has a background of

Neural Networks

There are several categories of statistical algorithms for data analysis within machine learning. Amongst them are neural networks, which have for the last decade exponentially been used within industry and academia for a number of usecases. From image analysis to weather prediction, these models are used extensively.

Neural networks, or feed forward neural networks (FFNN), are based on a few principles. First, the data is feeded forward through the network. The end output is evaluated in some fashion, and corrections are

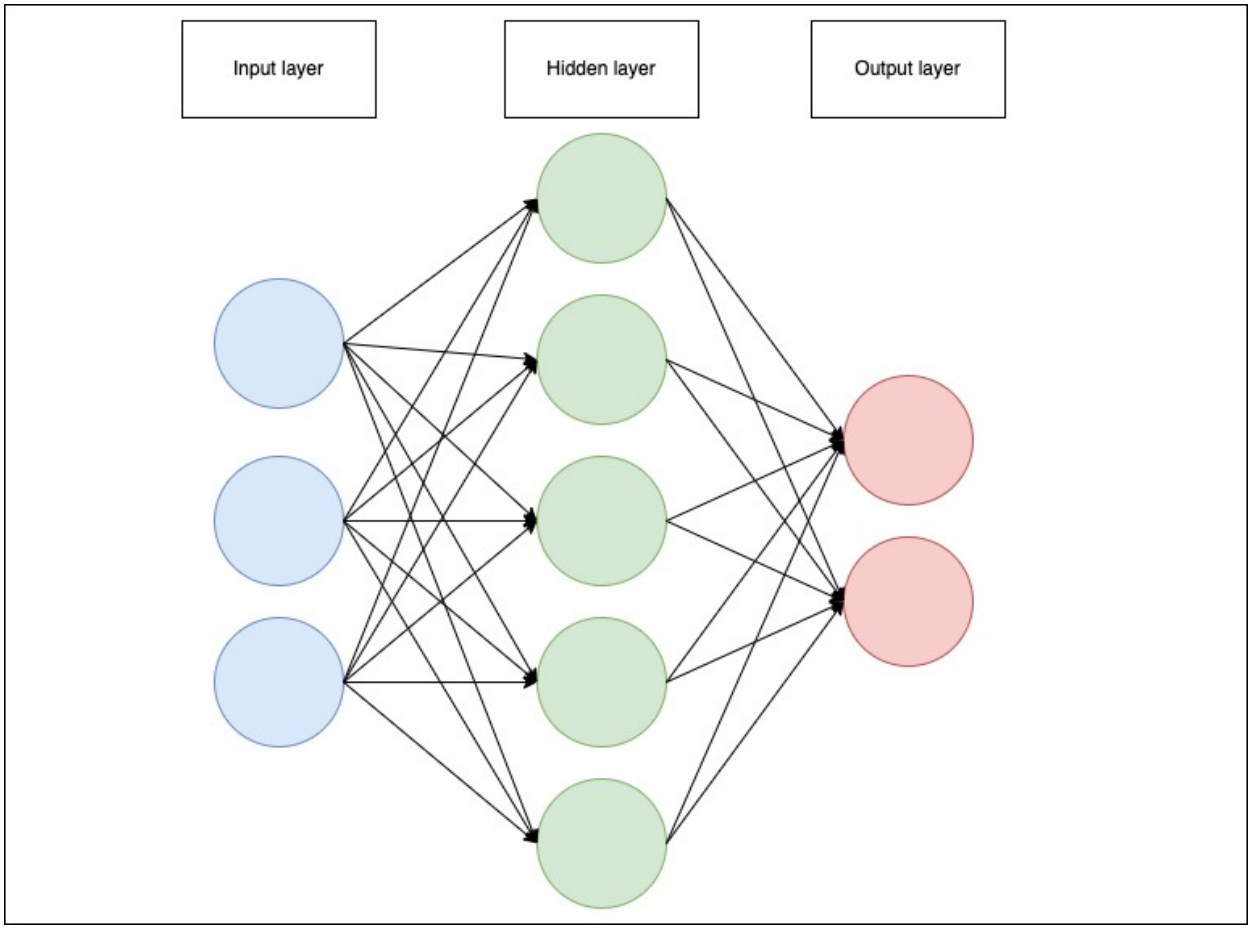


Figure 1.1: Simple neural network diagram drawm using Draw.io. Here the blue dots are the input layer, the green dots are a hidden layer, and the red dots are the output layer. The arrows shows the connections between each node.

then back propagated through the network, updating the weights and biases. This ”training” is done until a sufficient threshold is met. A general layout of a neural network is displayed in figure 1.1.

The input layer has the same shape of the dataset one uses to train or predict on, with one node for each feature in the dataset. The next layer is the hidden layers. For a given network, the amount of hidden layers can be tuned, as well as the number of nodes per layer. Finally, the last hidden layer is connected to the ouput layer, which is determined by the aim of the problem. In the case of figure 1.1, this neural network would represent a binary classification problem, in other words, two categories. The nodes in the network interacts through so called weights w and biases b. These are known as tunable parameters, which needs to be trained on the dataset before any prediction can be made.

In order to avoid confusion, we will adhere to table 1.1 for the notation used in the following sections.

Table 1.1: Notation

Matrices and vectors		
Notation	Description	Type
X	Design Matrix (input data).	$\mathbb{R}^{N\times\#\text{features}}$
t	Target values.	$\mathbb{R}^{N\times\#\text{categories}}$
y	Model output, the prediction from our network.	$\mathbb{R}^{N\times\#\text{categories}}$
W^l	The weight matrix associated with layer l which handles the connections between layer $l-1$ and l .	$\mathbb{R}^{n_{l-1}\times n_l}$
B^l	The bias vector associated with layer l which handles the biases for all nodes in layer l.	$\mathbb{R}^{n_l\times 1}$
Elements		
w_{ij}^l	The weight connecting node i in layer $l-1$ to node k in layer l.	$\mathbb{R}$
b_j^l	Bias acting on node j in layer l.	$\mathbb{R}$
z_j^l	Node output before activation on node j on layer l.	
a_j^l	Activated node output on node j on layer l.	$\mathbb{R}$
Functions		
C	Cost function	
σ^l	Activation function associated with layer l.	
Quantities		
n_l	The number of nodes in layer l.	
L	Number of layers in total with $L-2$ hidden layers.	
N	Total number of data points.	
All indexing starts from 1: $i, j, k, l = 1, 2, \ldots$		

Table 1.2: Table containing notation used for deriving the mathematical formulas for the neural network [2]

Gradient descent

Let us now consider a general n-dimansional problem, with parameters $\boldsymbol{\theta} = \{\theta_1, \theta_2, ..., \theta_n\}$. We want the set of $\boldsymbol{\theta}$ such that we minimize a costfunction with respect to the data and target. One way to solve this problem is using ordinary least squares. For this approach, the optimal paramters $\boldsymbol{\theta_{opt}}$ are derived from minimizing the cost function, as shown here:

$$\boldsymbol{\theta_{opt}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{t},$$

where $\boldsymbol{X}$ is the design matrix containing the data, and $\boldsymbol{t}$ is the target vector. This however leads to a problem. Suppose the design matrix is sufficiently large, then the matrix inversion will get computaitonally expensive,

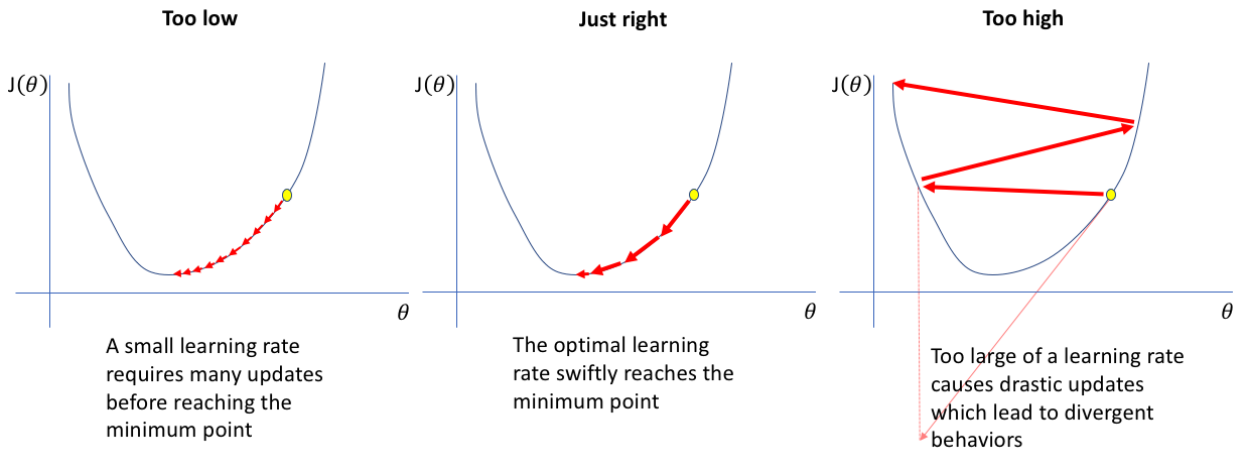


Figure 1.2: Figures showing different choice of learning rate for a given costfunction, with respect to the tunable parameters. Source: Jeremy Jordan, accessed 03.10.22.

or it might not even exist for a given $\boldsymbol{X}$. Thus, an alternative approch is to iteratively approximate the the ideal parameters.

Suppose we we have a cost function $C(\boldsymbol{\theta})$ for a given problem. We can approximate the minimum of the cost function by calculating the gradient $\nabla_\theta C$ with respect to $\boldsymbol{\theta}$. The negative of this gradient indicates the direction for the minimum of C when evaluating it in a specific point $\boldsymbol{\theta}_i$ in the parameter space [2]. This is formulized as follows

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \nabla_\theta C(\boldsymbol{\theta}_i), \tag{1.1}$$

where η is a step size, also called the learning rate. The choice of η is not a trivial case. It is one of severel hyperparameters[1] that can be altered, and that highly depend on the given problem. with regards to the learning rate, there are only three situations to consider, shown in figure 1.2.

Figure 1.2 visualizes the relation between the learning rate and the cost function. In the left most figure we note that the learning rate is too small. This leads to many iterations before you reach a minimum. In the right most figure we note that the learning rate is too high, and the result is that we get divergent behavior. Thus the goal is to find the optimal learning rate, shown in the middle figure.

A modified and prefered version of gradient descent is the so called stochastic gradient descent. Regular gradient descent can, for large datasets be quite slow, and is prone to getting stuck in local minima. To circumvent this issue, mini batches are introduced.

Feed forwarding

Inference (prediction) and training both use the same feed-forward algorithm. The procedure is to send the data through the network, weighting each connection according to the networks architecture, and produce an output. The procedure can be summarized in the following steps [2]:

- The data is recieved by the input nodes in teh network for each feature.
- Each input node weights the data value according to the connection of each node in the next layer.
- Every node in the hidden layers sums the weighted data values, and adds the bias associated to the given node, denoted as z.
- This value z is then sent through an activation function σ, which produces the output of the node, denoted as $a = \sigma(z)$.
- This process is repeated for each hidden layer, and it is important to note that the number of nodes in the hidden layers is not dependent on the number of features in the original dataset.
- The last hidden layer then sends the activated values to the output layer, where the number of nodes and choice of activation function depends on the problem to solve.

[1] Give reference to hyperparameters

Mathematically this is expressed as follows:

$$z_j^l = \sum_{i=1}^{n_{l-1}} w_{ij}^l a_i^{l-1} + b_j^l, \quad a_j^l = \sigma^l(z_j^l), \tag{1.2}$$

where l is the layer index, j is the node index, and i is the index of the node in the previous layer, and $l \neq 1$, as it is not used on the input layer.

Backpropagation

The way neural networks learn is conventionally by the use of the backpropagation algorithm, first proposed by Rumelhart et al[3]. This is a bit misleading, as the backprop algorithm actually only refers to how to compute the gradient[4]. The algorithm allows us to alter the weights and biases such that we get an ideal output. Assuming a costfunction C, we can calculate the gradient $\nabla_{w,b}C$, and use this to back propagate the error correction. The gradient $\nabla_{w,b}C$ is comprised of two derivatives:

Notethatthiscalculationisnotageneralizedalgorithmforbackpropagation, butratherforaspecialcaseusingMS

$$\nabla_{w,b}C = \left(\frac{\partial C}{\partial w_{i,j}^l}, \frac{\partial C}{\partial b_j^l} \right).$$

We have to use the chain rule to calculate the derivatives, and using that the last layer is $l = L$, we get the derivative with respect to the weights as

$$\frac{\partial C}{\partial w_{i,j}^L} = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial w_{i,j}^L},$$

where

$$a_j^L = \sigma(z_j^L), \quad z_j^L = \sum_{i=1}^{n_L - 1} w_{i,j}^L a_i^{L-1} + b_j^L.$$

This then gives us

$$\frac{\partial C}{\partial w_{i,j}^L} = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) a_i^{L-1}.$$

This derivative is very easy to calculate given a specific cost function and activation function. The derivative with respect to the bias is given as follows:

$$\frac{\partial C}{\partial b_k^L} = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial b_j^L},$$

which gives us the final expression as

$$\frac{\partial C}{\partial b_k^L} = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L).$$

We will now introduce a new notation, a local gradient commonly called the "error". It reflects how the rate of change of the cost function depends on the j'th node in the l'th layer.

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}.$$

Using this we get the following expression:

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L),$$

giving us the more compact forms of the derivatives with respect to the weights and biases:

$$\frac{\partial C}{\partial w_{i,j}^L} = \delta_j^L a_i^{L-1}, \quad \frac{\partial C}{\partial b_j^L} = \delta_j^L.$$

We can now let $\boldsymbol{\delta^l}$ be the vector of all the errors in the l'th layer, and $\boldsymbol{\delta^L}$ be the vector of all the errors in the last layer. The error in the l'th layer can then be expressed as a matrix equation for the last layer as follows:

$$\boldsymbol{\delta^l} = \nabla_a C \odot \frac{\partial \sigma}{\partial z^L}, \quad \nabla_a C = \left[\frac{\partial C}{\partial a_1^L}, \frac{\partial C}{\partial a_1^L}, ..., \frac{\partial C}{\partial a_{n_L}^L}\right]^T.$$

Here $\odot$ is the Hadamard product (element wise product). This local gradient can now be defined recursively for the j'th node in a layer l as a function of the local error in the next layer:

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}. \tag{1.3}$$

We also note that

$$z_k^{l+1} = \sum_{j=1}^{n_l} w_{j,k}^{l+1} a_j^l + b_k^{l+1} = \sum_{j=1}^{n_l} w_{j,k}^{l+1} \sigma(z_j^l) + b_k^{l+1},$$

thus the partial derivative is given as

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{j,k}^{l+1} \sigma'(z_j^l). \tag{1.4}$$

This allows us to substitute equation into equation to get the following expression:

$$\delta_j^l = \sum_k w_{j,k}^{l+1} \sigma'(z_j^l) \delta_k^{l+1}. \tag{1.5}$$

Using this we can derive a three step formula for the backprop algorithm:

- Compute the local gradient for the last layer, δ^L.
- Recursively compute the local gradient for the remaining layers, δ^l for $l = L-1, L-2, ..., 1$.
- Update the weights and biases for all layers, $l = 1, 2, ..., L.$ as shown below:

$$w_{i,j}^l \leftarrow w_{i,j}^l - \eta \delta_j^l a_i^{l-1},$$

$$b_j^l \leftarrow b_j^l - \eta \delta_j^l.$$

Autoencoders

Autoencoders are a subset of neural networks. Whereas a general neural network in principle can take any shape, autoencoders are more restrictive. This restrictiveness can in its most general sence we condensed into the following points:

- Same number of output categories as input categories
- A latent space with smaller dimensionality than the input/output layer

What we end up with two funnel shaped parts linked together. The two funnels are called the encoder (left funnel) and decoder (right funnel) respectively. This architecture is not accidental, but rather designed with a very specific solution of ploblems in mind, reconstruction. A good example to illustrate this is image denoising, illustrated in figure 1.3. Suppose you have a noised image, and want to denoise it. By feeding the encoder a noised image, and comparing the decoder output to the actual image, the autoencoder can tune itself to denoise images.

Mathematically this is is represented as follows. Using the annotations of each component in figure 1.3 we have that the decoded information is defined as follows

$$\mathbf{c} = \mathbf{E}_\phi(\mathbf{x}),$$

and the reconstruction given as

$$\mathbf{x}' = \mathbf{D}_\theta(\mathbf{E}_\phi(\mathbf{x})).$$

The parameters (ϕ, θ) are the tuneable parameters adjusted according to the loss function. In our case, the goal is reconstruction without copying, thus we can simply use mean squared error, given as

$$L_{AE}(\phi, \theta) = \frac{1}{N} \sum_{i=0}^{N-1} \left(\mathbf{x^i} - \mathbf{D}_\theta(\mathbf{E}_\phi(\mathbf{x^i}))\right)^2. \tag{1.6}$$

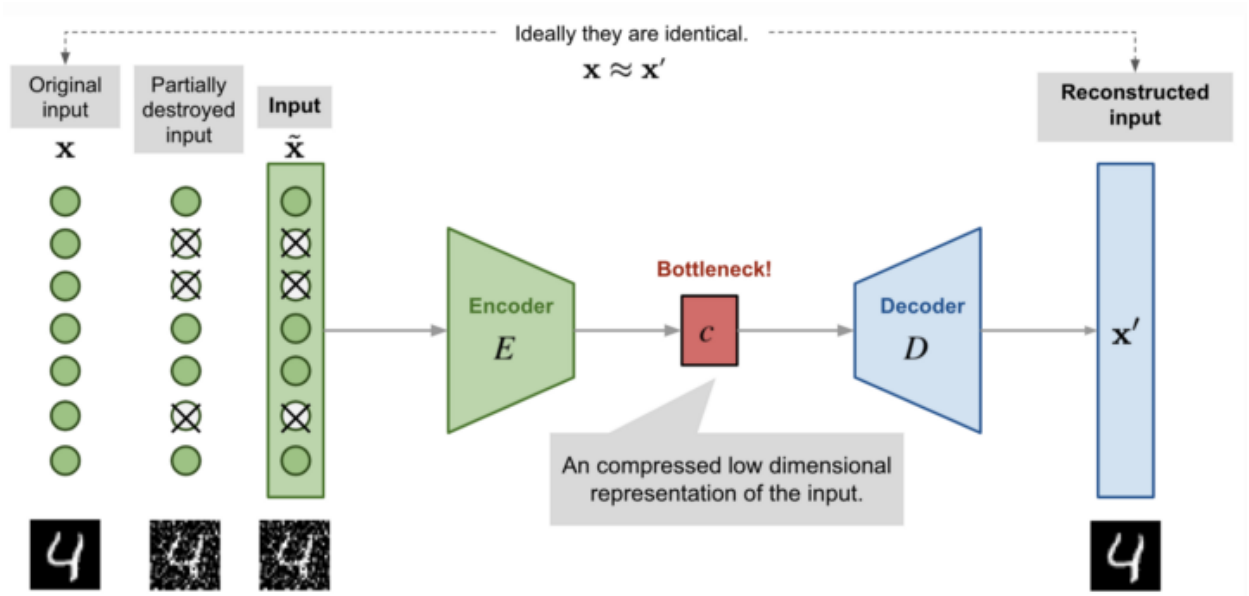


Figure 1.3: Figure depicting a model for an image denoising autoencoder. Here the input $\mathbf{x}$ is the original image, $\tilde{\mathbf{x}}$ is a noised version of $\mathbf{x}$, E is the encoder, D is the decoder, and c is the latent space. Found 27.09.22 here.

Optimizers

Optimizing [5],

Chapter 2

Standard model phenomenology

Structure and composition of the Standard Model

The standard model is to physicists what the periodic table is to chemists. It is comprised of two parent class particles, fermions and bosons, where fermions are comprised of quarks and leptons. The model contains 17 particles, 6 quarks, 6 leptons and 5 bosons, and are shown in figure 2.1.

Fermions

The fermions are the building blocks of matter, and contain two types of particles, leptons and quarks. Together they form protons and neutrons, atoms all around us. Fermions, unlike bosons, are spin half particles, and are also the only particles to have anti particles.

Quarks

Quarks are fractional charge particles, with defined charge of either $2/3$ or $-1/3$. They are the building blocks of protons and neutrons

Bosons

Limitations

All though the standard model have had great success comparing with experiments, there are still several problems not addressed by it. First and foremost, the standard model as described above, does not and cannot explain gravity in a quantized way. There are models that try to address this problem, but they supplement the standard model, and does not derrive it from it.

Proposal model

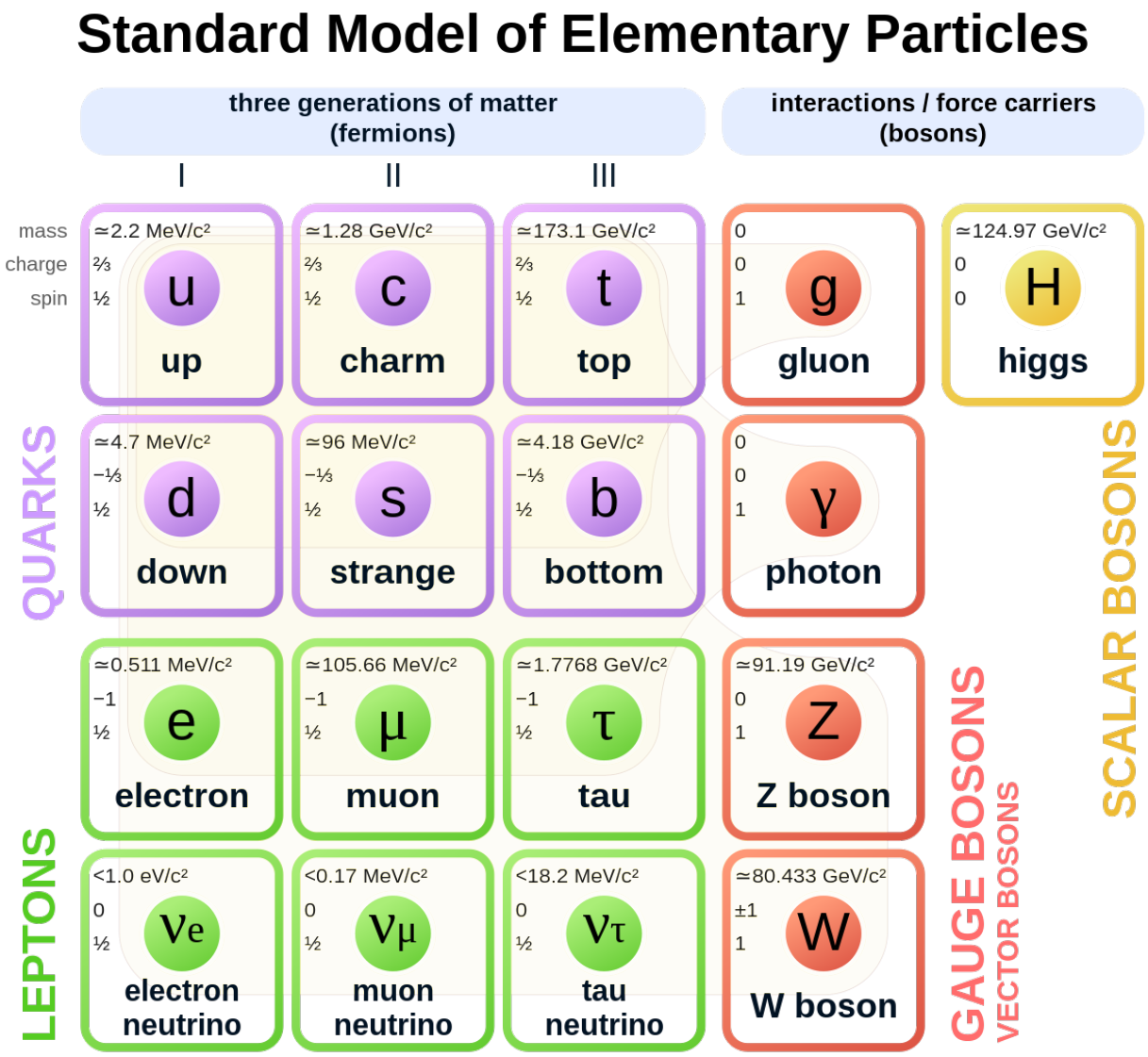


Figure 2.1: The standard model of elementary particles. Source here. Accessed 07.10.22

Chapter 3

Implementation

ATLAS

, [6], [7]

Data collection

The ATLAS detector three selection stages before the data is stored. In order to reach the highest intensity of collisions, the LHC accellerates packets of around 10^{11} protons, and collides them at a rate of 25 nanoseconds, yeilding a collision rate of 40 MHz[8]. [9]

Triggers

Data preparation

ROOT

It is a lot. [10]

ROOT, memory management etc.

N tuples

The main datastructure of ROOT is the so called N tuple structure. This datastructure contains each property for each type of particle in a given event, yeilding a ragged structure.

RDataFrame

[11]

RDataFrame's main purpose is to make reading and handling of root files easier, especially in relation to modern machine learning tools and their framework. This is done by creating a dataframe like structure of the root n-tuples, and then lazily apply contraints to the data. Using PyROOT, RDataFrame can be accessed in Python, as the functionality is wrapped around a C++ class. Below is an example of how to create a RDataFrame object, apply a cut and then create a column for later use. Here, good leptons are defined first, denoted as "ele_SG" and "muo_SG". A cut is then applied where we require that the number of good leptons is always 3. Finally, a column is created where the combination of type of leptons in the 3 lepton system is stored, as well as creating a histogram containing the results for that given channel k. Notice here that if the variable already exist as a column in the dataframe, arithmetic and logic can be

[1] In this context lazily means that the functions and or cuts are done first after all have been registered, see ROOT guidelines for more.

	$jetP_T$	$jetPhi$	$lepP_T$	$lepPhi$
0	$[120.2, 57]$	$[1.2, 0.5]$	$[223.3, 57.5, 9.7]$	$[0.545, 0.2, -0.3]$
1	$[,]$	$[,]$	$[121.343, 89.323]$	$[0.886, -0.855]$
2	$[86.112]$	$[86.112]$	$[57.75, 34.5]$	$[0.33, 0.255]$

Rowlength
10
4
6

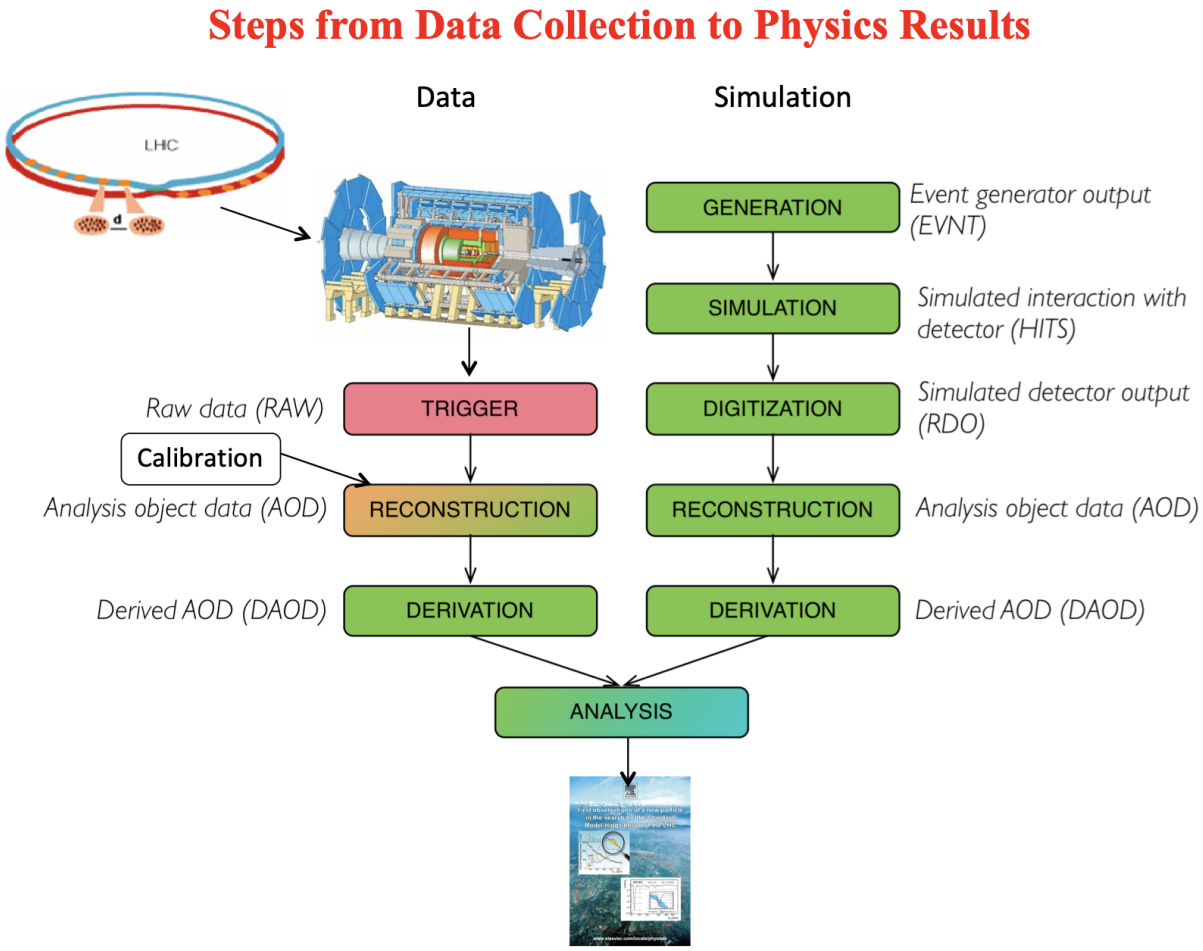


Figure 3.1: Figure describing the steps to take for data collection at ATLAS, fetched from Hybrid ATLAS Induction Day + Software Tutorial workshop, part Computing and Data preparation, held by S.M Wang [8] .

applied directly using those columns to create new one. More complicated information, such as the flavor combination for the leptons, or the invariant mass of two particles must be found or calculated using C++ functions. An example of such a function is shown below the python code listing.

```python
import ROOT as R

R.EnableImplicitMT(200)
R.gROOT.ProcessLine(".L helperFunctions.cxx+")
R.gSystem.AddDynamicPath(str(dynamic_path))
R.gInterpreter.Declare(
    '#include "helperFunctions.h"'
)  # Header with the definition of the myFilter function
R.gSystem.Load("helperFunctions_cxx.so")  # Library with the myFilter function

df_mc = getDataFrames(mypath_mc)
df_data = getDataFrames(mypath_data)
df = {**df_mc, **df_data}

for k in df.keys():

    # Signal leptons
    df[k] = df[k].Define(
        "ele_SG",
        "ele_BL && lepIsoLoose_VarRad && lepTight && (lepD0Sig <= 5 && lepD0Sig >= -5)",
    )
    df[k] = df[k].Define(
        "muo_SG",
        "muo_BL && lepIsoLoose_VarRad && (lepD0Sig <= 3 && lepD0Sig >= -3)",
    )
    df[k] = df[k].Define("isGoodLep", "ele_SG || muo_SG")
    df[k] = df[k].Define(
                "nlep_SG", "ROOT::VecOps::Sum(ele_SG)+ROOT::VecOps::Sum(muo_SG)"
            )

    df[k] = df[k].Filter("nlep_SG == 3", "3 SG leptons")

    # Define flavor combination based on
    df[k] = df[k].Define("flcomp", "flavourComp3L(lepFlavor[ele_SG || muo_SG])")
    histo[f"flcomp_{k}" ] = df[k].Histo1D(
        (
            f"h_flcomp_{k}",
            f"h_flcomp_{k}",
            len(fldic.keys()),
            0,
            len(fldic.keys()),
        ),
        "flcomp",
        "wgt_SG",
    )

```

```cpp
double getM(VecF_t &pt_i, VecF_t &eta_i, VecF_t &phi_i, VecF_t &e_i,
            VecF_t &pt_j, VecF_t &eta_j, VecF_t &phi_j, VecF_t &e_j,
            int i, int j)
{
/* Gets he invariant mass between two particles, be it jets or leptons */

const auto size_i = int(pt_i.size());
const auto size_j = int(pt_j.size());

if (size_i == 0 || size_j == 0){return 0.;}
if (i > size_i-1){return 0.;}
if (j > size_j-1){return 0.;}

TLorentzVector p1;
TLorentzVector p2;

p1.SetPtEtaPhiM(pt_i[i], eta_i[i], phi_i[i], e_i[i]);
p2.SetPtEtaPhiM(pt_j[j], eta_j[j], phi_j[j], e_j[j]);

double inv_mass = (p1 + p2).M();

return inv_mass;
}
```

Using ROOT we can create Lorentzvectors to calculate a number of properties, such as the invariant mass of two particles.

```
import pandas as pd

cols = df.keys()

for k in cols:

    print(f"Transforming {k}.ROOT to numpy")
    numpy = df[k].AsNumpy(DATAFRAME_COLS)
    print(f"Numpy conversion done for {k}.ROOT")
    df1 = pd.DataFrame(data=numpy)
    print(f"Transformation done")


    df1.to_hdf(
        PATH_TO_STORE + f"/{k}_3lep_df_forML_bkg_signal_fromRDF.hdf5", "mini"
    )
```

Once a dataframe has been created, the columns chosen and histograms are drawn, the dataframe can be converted to a pandas dataframe, which is a very popular choice for data structure when doing data analysis in python. Here the new pandas dataframe is stored as hdf5[12] files, for later use.

The dataset features

RMM matrix

Most of the features in the analysis are elements in the so called Rapidity-Mass (RMM) matrix inspired by the work of Chekanov [13].

!! Motivation for using such a matrix in machine learning $\rightarrow$ hint to highly uncorrolated feats!!

Its composition is determined as a square matrix of $1 + \sum_{i=1}^{T} N_i$ columns and rows, where T is the total number of objects (i.e jets, electrons etc.), and N_i is the multiplicity of a given object. In the case of the same number of a given object for all objects, we can denote the RMM matrix as a TmNn matrix, where m is the multiplicity of T, and n is the number of particle per type. Thus there is already room for evaluation, as the combination of number of objects and the number of each object type highly affects the analysis as well as computational resources. Each cell in the matrix contains information about either single og two particle properties. An example is shown in matrix 3.1.

$$\begin{pmatrix} \boldsymbol{e}_T^{miss} & m_T(j_1) & m_T(j_2) & m_T(e_1) & m_T(e_2) \\ h_L(j_1) & \boldsymbol{e_T}(j_1) & m(j_1,j_2) & m(j_1,e_1) & m(j_1,e_2) \\ h_L(j_2) & h(j_2,j_1) & \delta\boldsymbol{e_T}(j_2) & m(j_2,e_1) & m(j_2,e_2) \\ h_L(e_1) & h(e_1,j_1) & h(e_1,j_2) & \boldsymbol{e_T}(e_1) & m(e_1,e_2) \\ h_L(e_2) & h(e_2,j_1) & h(e_2,j_2) & h(e_2,e_1) & \delta\boldsymbol{e_T}(j_2) \end{pmatrix} \tag{3.1}$$

In matrix 3.1 we have the RMM matrix for a T2N2 system, in other words we have two types of particles, jets and electrons, where each type has two particles. The matrix itself is partitioned into three parts. The diagonal represents energy properties, the upper triangular represents mass properties, and the lower triangular represents longitudal properties related to rapidity. The diagonal has three different properties, $\boldsymbol{e_T^{miss}}$, $\boldsymbol{e_T}$ and $\delta\boldsymbol{e_T}$. $\boldsymbol{e_T^{miss}}$ is placed in the $(0,0)$ in the matrix. It accounts for the missing energy for the system, which is of high interest for this analysis due to the search for heavy neutrinos. $\boldsymbol{e_T}$ is the transverse energy defined as

$$\boldsymbol{e_T} = \sqrt{m^2 + p_T^2}$$

but for light particles such as electrons, this can be approximated to $\boldsymbol{e_T} \approx p_T$. $\delta\boldsymbol{e_T}$ is the transverse energy imbalance. It is defined as

$$\delta\boldsymbol{e_T} = \frac{E_T(i_n - 1) - E_T(i_n)}{E_T(i_n - 1) + E_T(i_n)},\ n = 2, ..., N.$$

The first columm in the RMM matrix, with the exeption of the first element, is related to the longitudal property of the given particle. It is defined as

$$h_L(i_n) = C(\cosh(y) - 1),$$

where C is a constant to ensure that the average $h_L(i_n)$ values do not deviate too much from the ranges of the invariant masses of the transverse masses, found to be 0.15[13]. y is the rapidity of the particle, and i_n is

the particle number. On the lower triangle we have the longitudal properties of the combinations of particles. Similar to $h_L(i_n)$, this property is defined as

$$h(i_n, j_k) = C(\cosh(\Delta y) - 1),$$

where $\Delta y = y_{i_n} - y_{j_k}$ is the rapidity difference for particle i_n and j_k.

MonteCarlo and data comparison

Before we can start the analysis, we need to compare the MonteCarlo and data. This is done to ensure that the training samples we use are actually useful. As described by R. Stuart Geiger et al. [14], the concept of "Garbage in, garbage out" is of key importance in computer science, and indeed important in high energy physics. To ensure that the models we train actually learns physical processes, the training set must represent the physics "status quo". If the training samples do not match the physical reality, we regard it, in the context of high energy physics, as garbage in, which will in turn give garbage out. The Monte Carlo standard model simulations are indeed very good, but the are numerical approximations, and can sometimes be off. Thus, every feature that well be used for training have to be checked before being used. This is done by comparing the distributions of the features in the MonteCarlo and ATLAS data. Assuming that there are no issues with the data collection from ATLAS, we then assume that the data from ATLAS is the "ground truth", as it is the only dataset we can analyze, and the only dataset we can compare to. The MonteCarlo is then compared to the data, and if the distributions are similar, we can assume that the training samples are good enough.

In figure 3.2 two features have been selected to vizualize the comparison between Monte Carlo and ATLAS data, e_T^{miss} and $flcomp$. We see that both e_T^{miss} and $flcomp$ satisfy a good ratio between Monte Carlo and ATLAS data, thus we can safely move forward with the analysis. All features were checked, and can be found in appendix B.

Tabular data

Code implementation

The machine learning analysis was written with Keras[15] using the Tensorflow api[16] . The machine learning structure was written using a functional structure[2] In practise, this model could just as well have been written as a Sequential model[3], but at a cost of flexibility and lack of potential non-linear structure in the architecture.

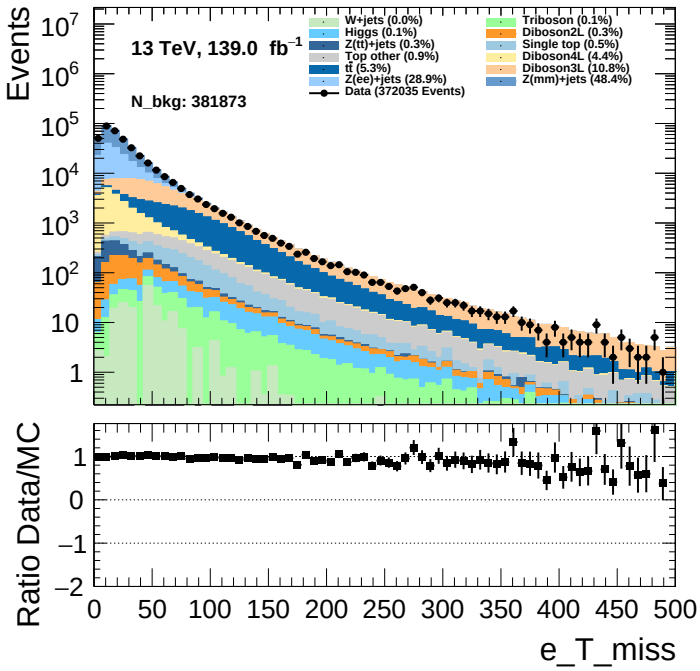
Construction of a neural network in Tensorflow

Using the functional structure, a general neural network in the Tensorflow API can be constructed as shown below.

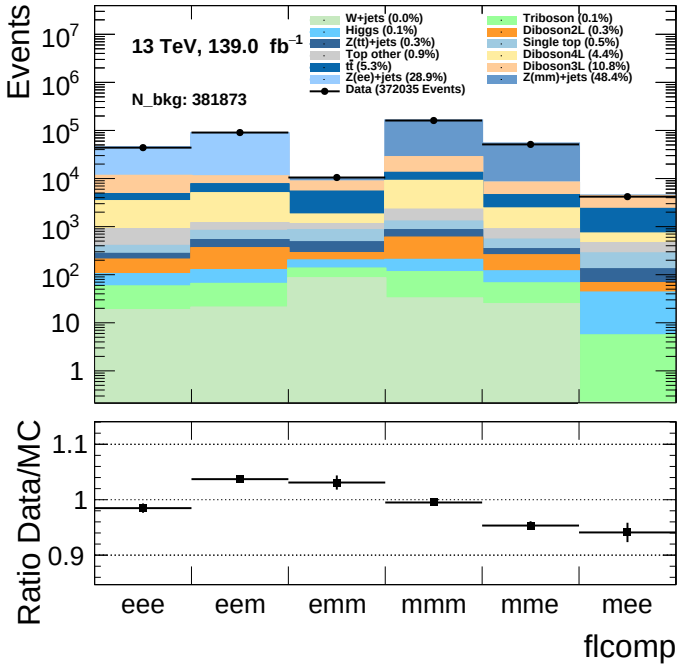
```
import tensorflow as tf


inputs = tf.keras.layers.Input(shape=data_shape, name="input")

# First hidden layer
First_layer = tf.keras.layers.Dense(
    units=30,
    activation="relu"
)(inputs)

# Second hidden layer
Second_layer = tf.keras.layers.Dense(
    units=45,
    activation="relu"
)(First_layer)

# Second hidden layer
output_layer = tf.keras.layers.Dense(
    units=1,
```

[2]Functional structure uses a function call for layers, i.e for layers a,b, then b(a) will connect the two layers, and equals a sequential link a → b. This allows for more flexible structures. More on the functional api can be found here.

[3]Sequential structure adds layers in sequence, i.e for layers a, b, c we have that a → b → c, with a strict structure. This allows for more organized code. More on sequential models can be found here.



(a) Missing transverse energy for the three lepton final state. The histogram contains the entire Run 2 dataset.



(b) Flavor combination for the three lepton final state. This histogram only contains the flavor combinations for the good leptons, denoted lep_{SG}. The histogram contains the entire Run 2 dataset.

Figure 3.2: Comparison of the MonteCarlo and data for the three lepton final state with the features e_T^{miss} and flavor composition.

```
    activation="sigmoid"
)(Second_layer)


# Model definition
nn_model = tf.keras.Model(inputs, output_layer, name="nn_model")

hp_learning_rate = 0.0015
optimizer = tf.keras.optimizers.Adam(hp_learning_rate)
nn_model.compile(loss="mse", optimizer=optimizer, metrics=["mse"])
```

The neural network here contains one input layer, two hidden layers, and an output layer. The choice of nodes and activation functions are arbitrary here as the use case has not been defined. Note that this is exactly the same as this example, using a sequential structure.

```
import tensorflow as tf

nn_model = tf.keras.Sequential(
    [
        tf.keras.layers.Dense(30, activation="relu", input_shape=data_shape),
        tf.keras.layers.Dense(45, activation="relu"),
        tf.keras.layers.Dense(1, activation="sigmoid"),
    ]
)

hp_learning_rate = 0.0015
optimizer = tf.keras.optimizers.Adam(hp_learning_rate)
nn_model.compile(loss="mse", optimizer=optimizer, metrics=["mse"])
```

In the case of this project, we want to create an autoencoder, as shown in figure 1.3. A general network archtitecture is proposed below, where the nodes, activation functions, and other hyperparameters can be tuned.

```
import tensorflow as tf

class HyperParameterTuning(RunAE):
    def __init__(self, data_structure: object, path: str)->None:
        super().__init__(data_structure, path)

    def AE_model_builder(self, hp: kt.engine.hyperparameters.HyperParameters):

        """_summary_

        Args:
            hp (kt.engine.hyperparameters.HyperParameters): _description_

        Returns:
            _type_: _description_
        """
        ker_choice = hp.Choice("Kernel_reg", values=[0.5, 0.1, 0.05, 0.01])
        act_choice = hp.Choice("Atc_reg", values=[0.5, 0.1, 0.05, 0.01])

        alpha_choice = hp.Choice("alpha", values=[1.0, 0.5, 0.1, 0.05, 0.01])

        # Activation functions
        activations = {
            "relu": tf.nn.relu,
            "tanh": tf.nn.tanh,
            "leakyrelu": "leaky_relu",
            "linear": tf.keras.activations.linear,
        }  # lambda x: tf.nn.leaky_relu(x, alpha=alpha_choice),

        # Input layer
        inputs = tf.keras.layers.Input(shape=self.data_shape, name="encoder_input")

        # First hidden layer
        x = tf.keras.layers.Dense(
            units=hp.Int(
                "num_of_neurons1", min_value=60, max_value=self.data_shape - 1, step=1
            ),
            activation=activations.get(
                hp.Choice("1_act", ["relu", "tanh", "leakyrelu", "linear"])
            ),
            kernel_regularizer=tf.keras.regularizers.L1(ker_choice),
            activity_regularizer=tf.keras.regularizers.L2(act_choice),
```

```
        )(inputs)

        # Second hidden layer
        x_ = tf.keras.layers.Dense(
            units=hp.Int("num_of_neurons2", min_value=30, max_value=59, step=1),
            activation=activations.get(
                hp.Choice("2_act", ["relu", "tanh", "leakyrelu", "linear"])
            ),
        )(x)

        # Third hidden layer
        x1 = tf.keras.layers.Dense(
            units=hp.Int("num_of_neurons3", min_value=10, max_value=29, step=1),
            activation=activations.get(
                hp.Choice("3_act", ["relu", "tanh", "leakyrelu", "linear"])
            ),
            kernel_regularizer=tf.keras.regularizers.L1(ker_choice),
            activity_regularizer=tf.keras.regularizers.L2(act_choice),
        )(x_)

        val = hp.Int("lat_num", min_value=1, max_value=9, step=1)

        # Forth hidden layer
        x2 = tf.keras.layers.Dense(
            units=val,
            activation=activations.get(
                hp.Choice("4_act", ["relu", "tanh", "leakyrelu", "linear"])
            ),
        )(x1)

        # Encoder definition
        encoder = tf.keras.Model(inputs, x2, name="encoder")

        # Latent space
        latent_input = tf.keras.layers.Input(shape=val, name="decoder_input")

        # Fifth hidden layer
        x = tf.keras.layers.Dense(
            units=hp.Int("num_of_neurons5", min_value=10, max_value=29, step=1),
            activation=activations.get(
                hp.Choice("5_act", ["relu", "tanh", "leakyrelu", "linear"])
            ),
            kernel_regularizer=tf.keras.regularizers.L1(ker_choice),
            activity_regularizer=tf.keras.regularizers.L2(act_choice),
        )(latent_input)

        # Sixth hidden layer
        x_ = tf.keras.layers.Dense(
            units=hp.Int("num_of_neurons6", min_value=30, max_value=59, step=1),
            activation=activations.get(
                hp.Choice("6_act", ["relu", "tanh", "leakyrelu", "linear"])
            ),
        )(x)

        # Seventh hidden layer
        x1 = tf.keras.layers.Dense(
            units=hp.Int(
                "num_of_neurons7", min_value=60, max_value=self.data_shape - 1, step=1
            ),
            activation=activations.get(
                hp.Choice("7_act", ["relu", "tanh", "leakyrelu", "linear"])
            ),
            kernel_regularizer=tf.keras.regularizers.L1(ker_choice),
            activity_regularizer=tf.keras.regularizers.L2(act_choice),
        )(x_)

        # Output layer
        output = tf.keras.layers.Dense(
            self.data_shape,
            activation=activations.get(
                hp.Choice("8_act", ["relu", "tanh", "leakyrelu", "linear"])
            ),
        )(x1)

        # Encoder definition
```

```
118         decoder = tf.keras.Model(latent_input, output, name="decoder")
119 
120         # Output definition
121         outputs = decoder(encoder(inputs))
122 
123         # Model definition
124         AE_model = tf.keras.Model(inputs, outputs, name="AE_model")
125 
126         hp_learning_rate = hp.Choice(
127             "learning_rate", values=[9e-2, 9.5e-2, 1e-3, 1.5e-3]
128         )
129         optimizer = tf.keras.optimizers.Adam(hp_learning_rate)
130 
131         AE_model.compile(loss="mse", optimizer=optimizer, metrics=["mse"])
132 
133         return AE_model
134 
```

This function creates a tensorflow.keras model that has tuneable structure, thus allowing for optimized tuning with the Keras-Tuner library[17].

Chapter 4

Results

Chapter 5

Discussion

Conclusion

Future work, more work

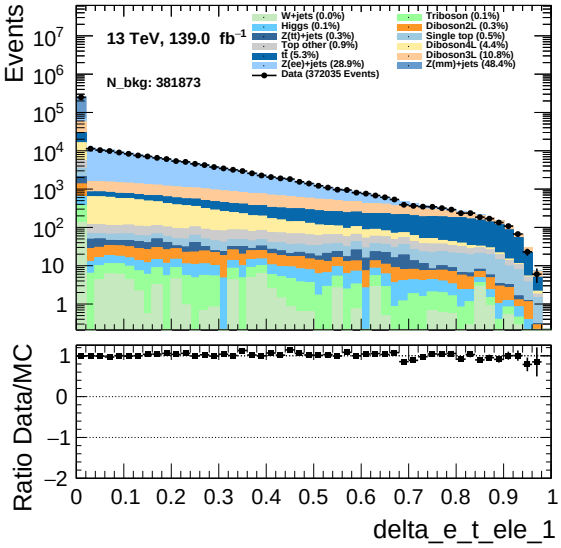
Appendices

Appendix A

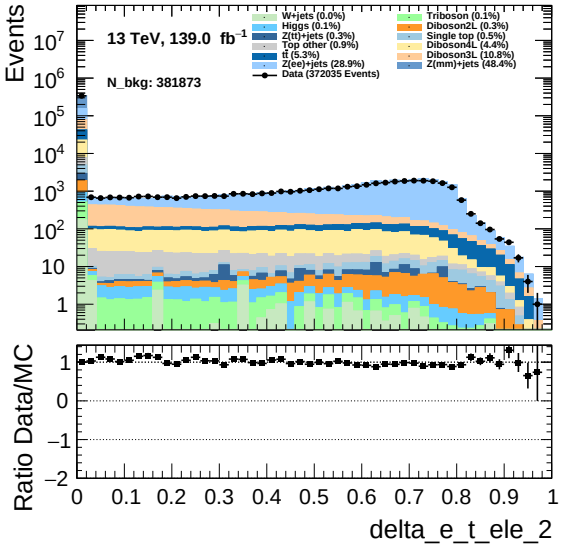
Appendix B

Feature checks between Monte Carlo and ATLAS data

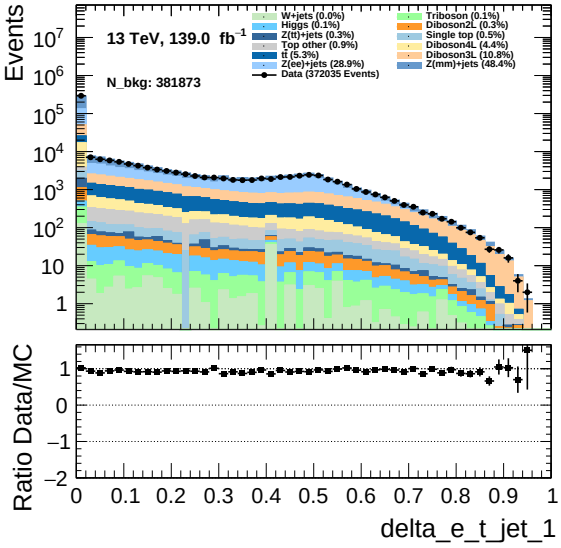
The figures below are made smaller to reduce the amount of pages. Note that the legend for each figure is the same, thus, for rerence on the legend, see figure 3.2 in subsection 3 on Monte Carlo and ATLAS data comparison.



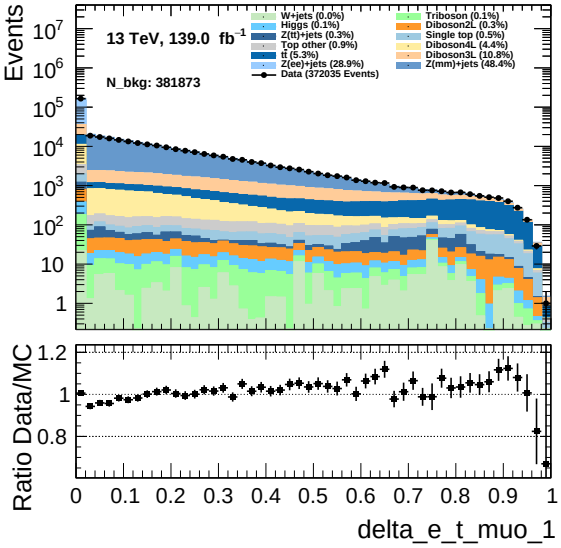
(a) Transverse energy imbalance for the second electron.



(b) Transverse energy imbalance for the third electron.

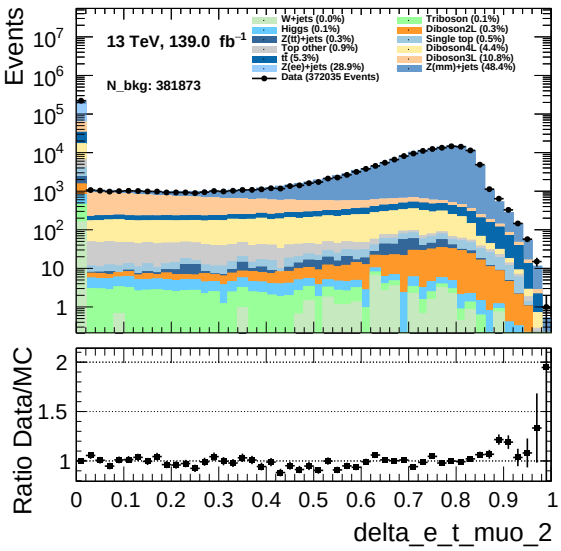


(c) Transverse energy imbalance for the second jet.

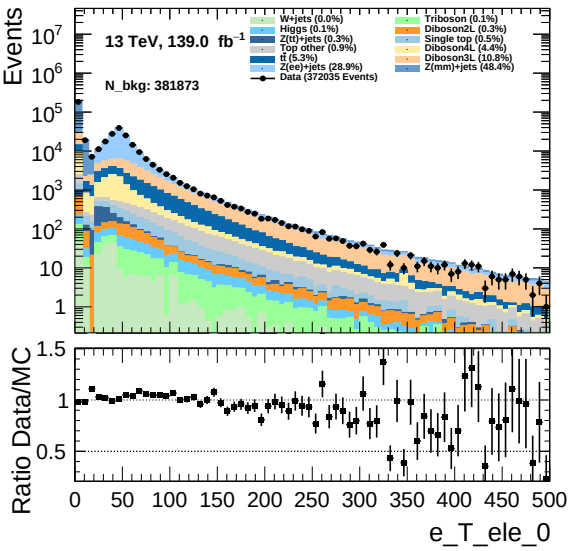


(d) Transverse energy imbalance for the second muon.

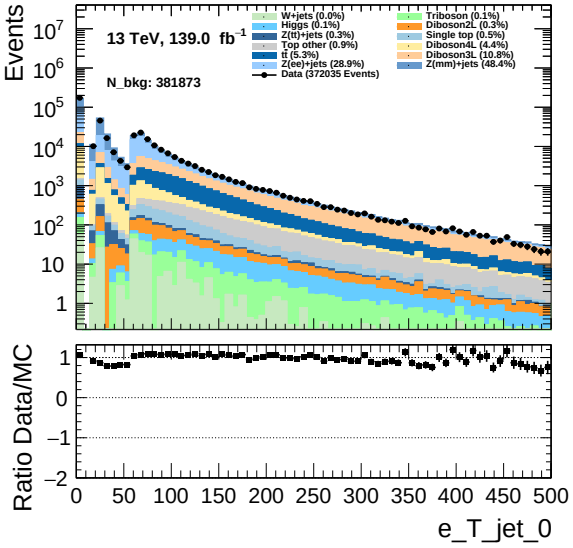
Figure 1: The four figures in this plot contains the transverse energy imbalance for the second muon and jet as well as the second and third electron.



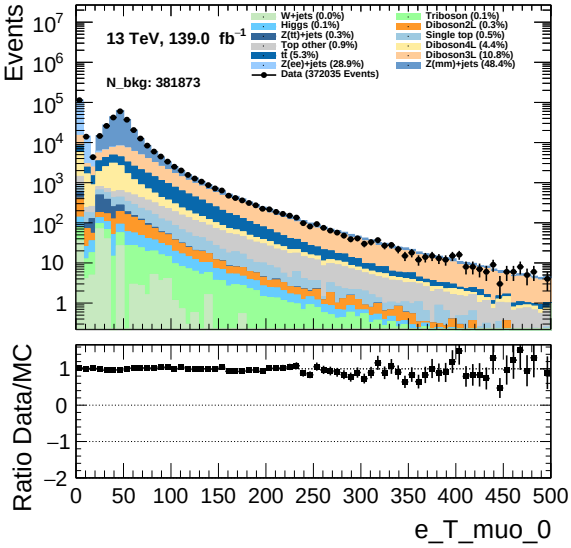
(a) Transverse energy imbalance for the third muon.



(b) Transverse energy for the first electron.

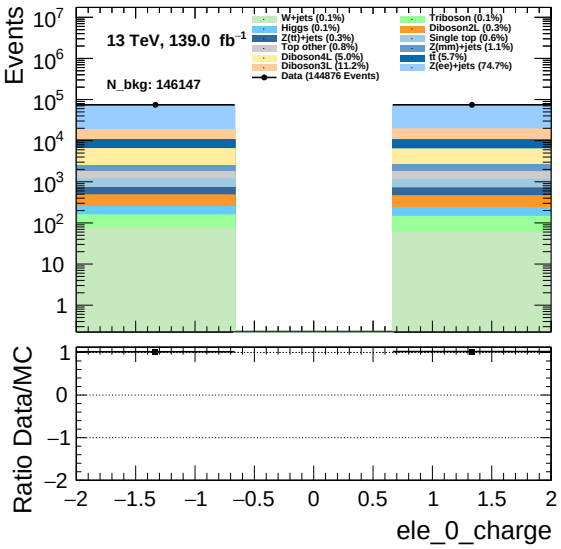


(c) Transverse energy for the first jet.

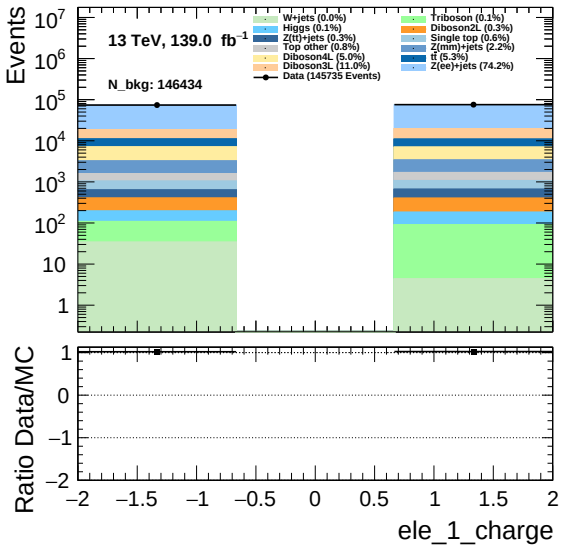


(d) Transverse energy for the first moun.

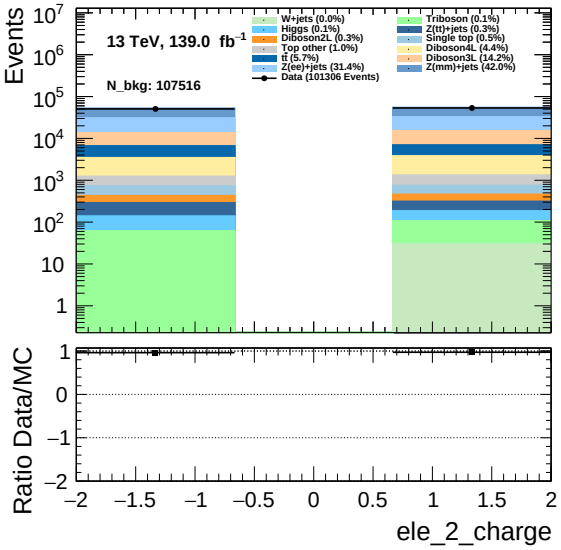
Figure 2: Transverse energy for the first jet, electron and muon, as well as the transverse energy imbalance for the third muon.



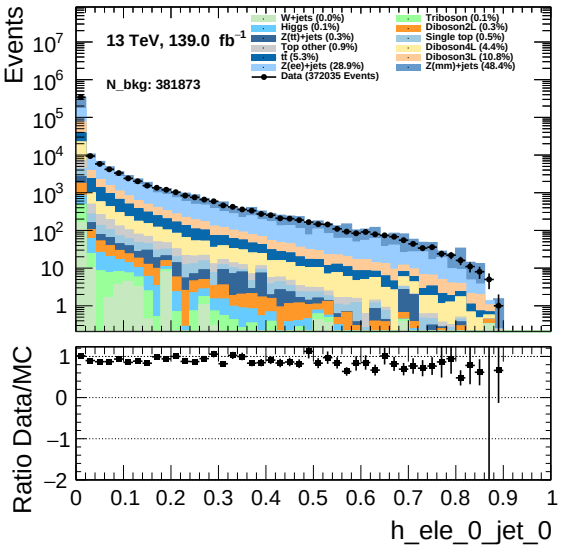
(a) Change distribution for the first electron.



(b) Change distribution for the second electron.

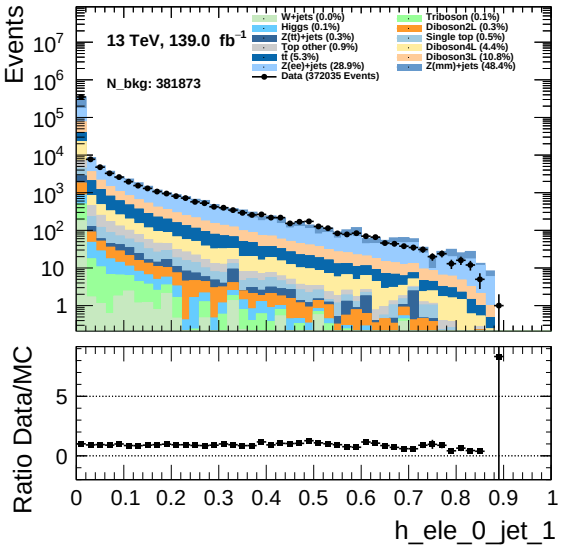


(c) Change distribution for the third electron.

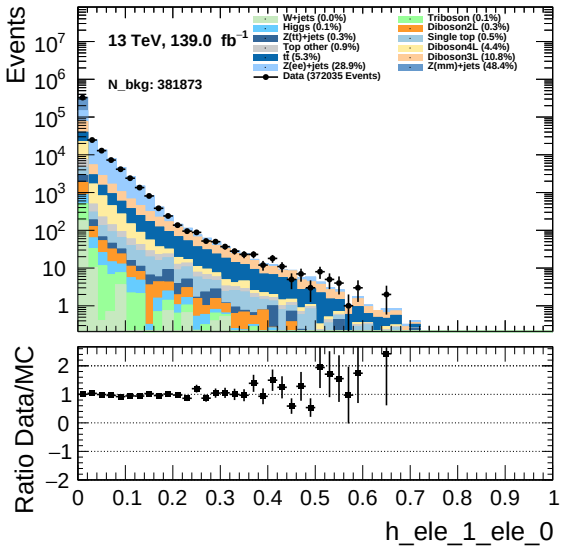


(d) Longitudal component basd on rapidity difference between the first electron and first jet.

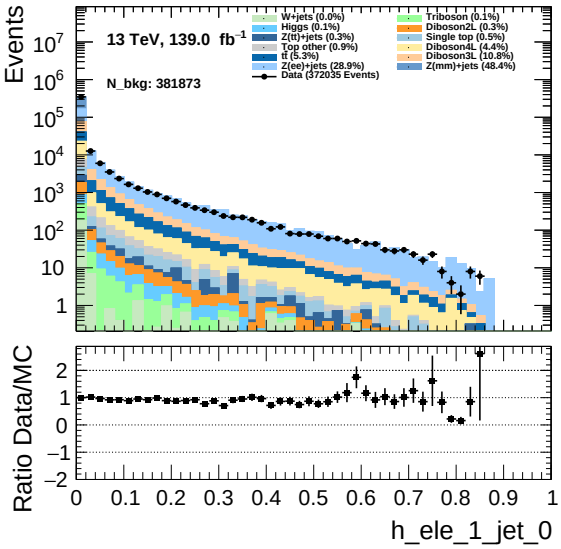
Figure 3: Charge distribution for the three electrons, and longitudal component between first electron and first jet.



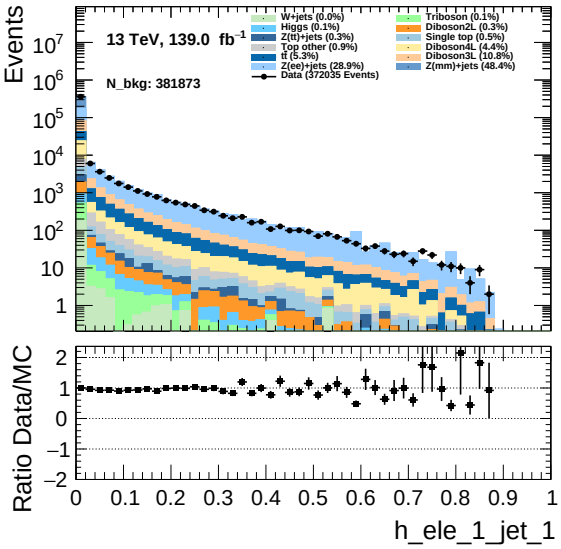
(a) Longitudal component between first electron and second jet.



(b) Longitudal component between first and second electron.

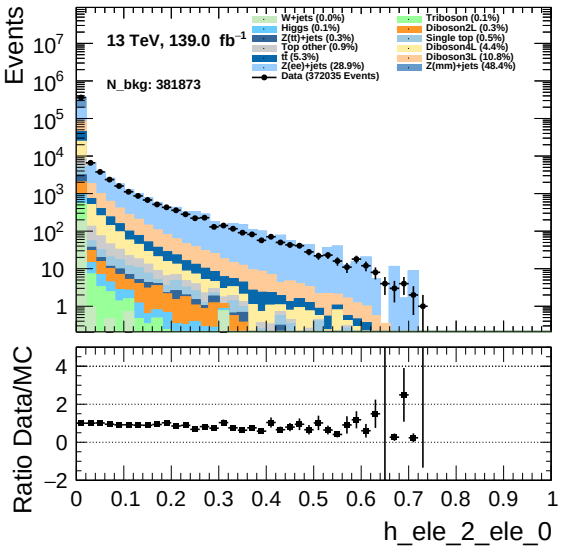


(c) Longitudal component between second electron and first jet.

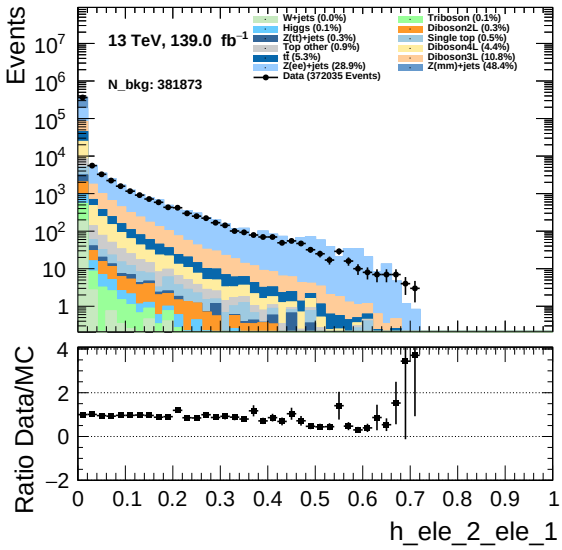


(d) Longitudal component between second electron and second jet.

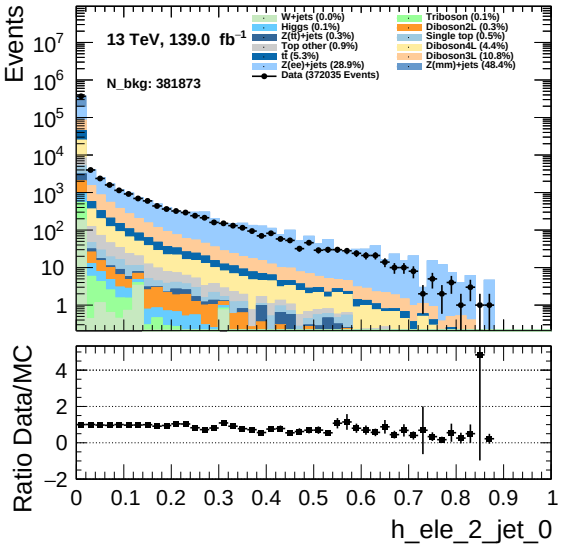
Figure 4: Longitudal components between electrons and electrons, and electrons and jets.



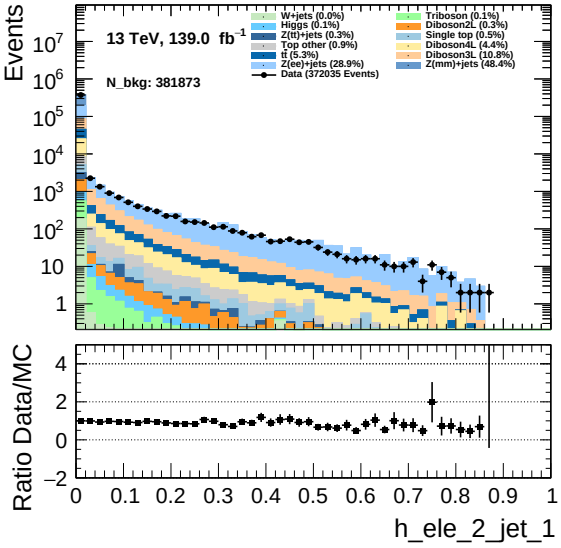
(a) Longitudal component between first and third electron.



(b) Longitudal component between second and third electron.

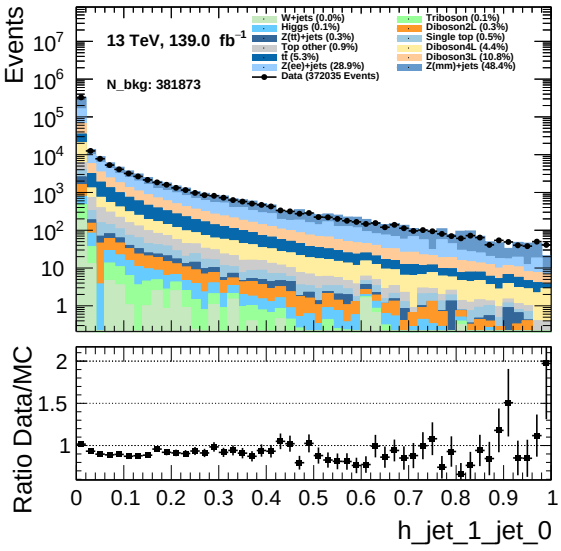


(c) Longitudal component between third electron and first jet.

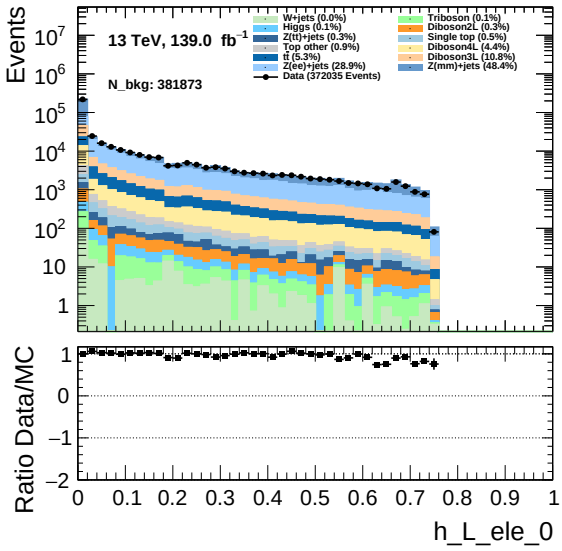


(d) Longitudal component between third electron and second jet.

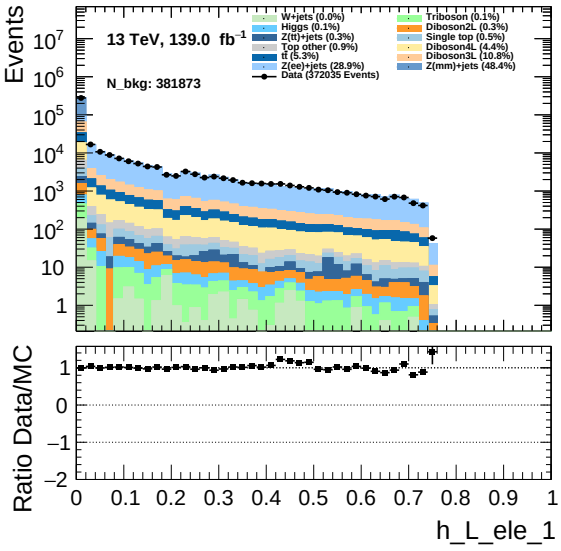
Figure 5: Longitudal component between between electrons and electrons, and electrons and jets.



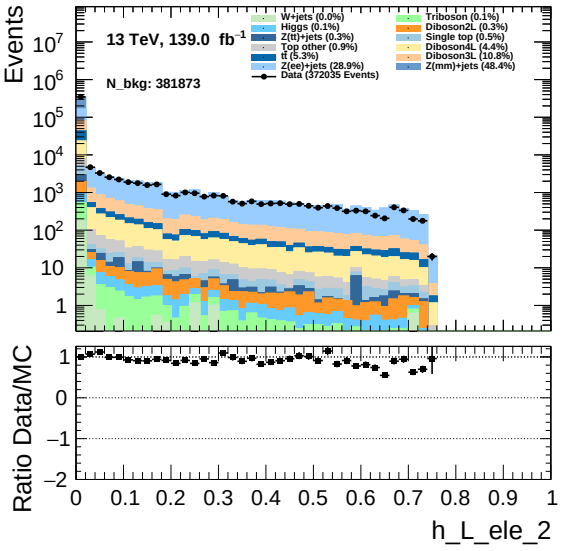
(a) Longitudal component between first and second jet.



(b) Longitudal component between for first electron

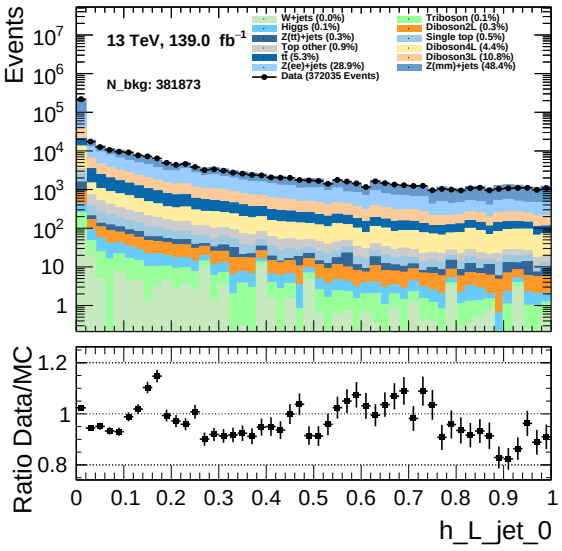


(c) Longitudal component for second electron.

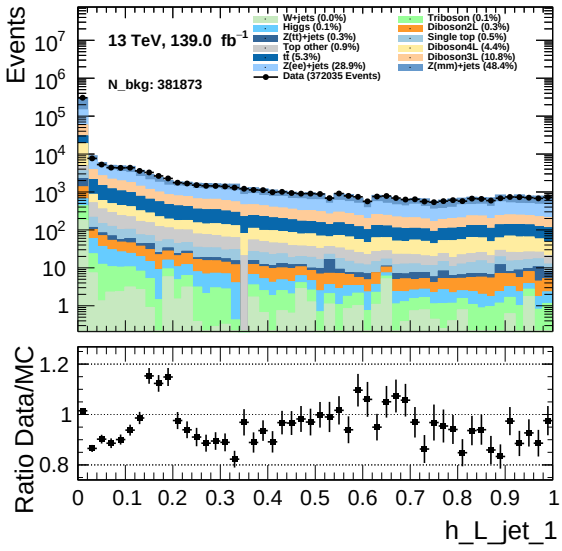


(d) Longitudal component for third electron.

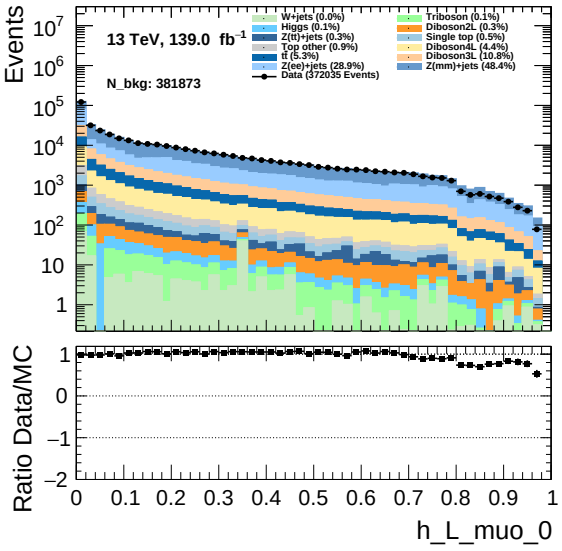
Figure 6: Longitudal component between electron and jet, and for individual electrons.



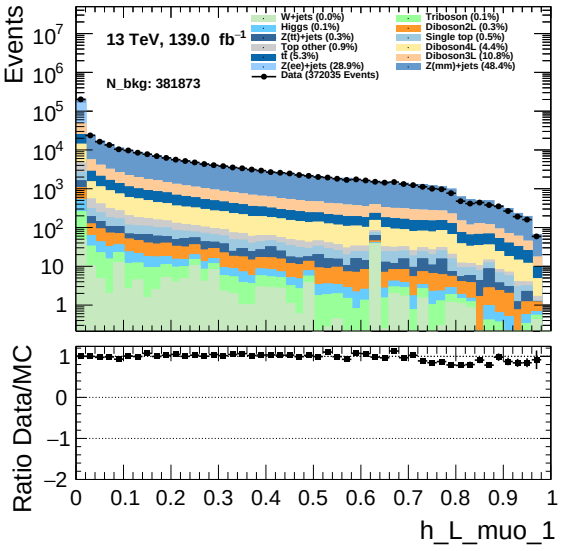
(a) Longitudal component for the first jet.



(b) Longitudal component for the second jet.

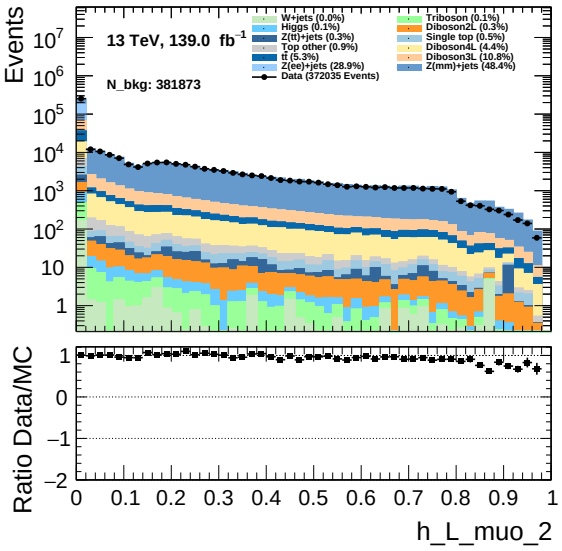


(c) Longitudal component for the first muon.

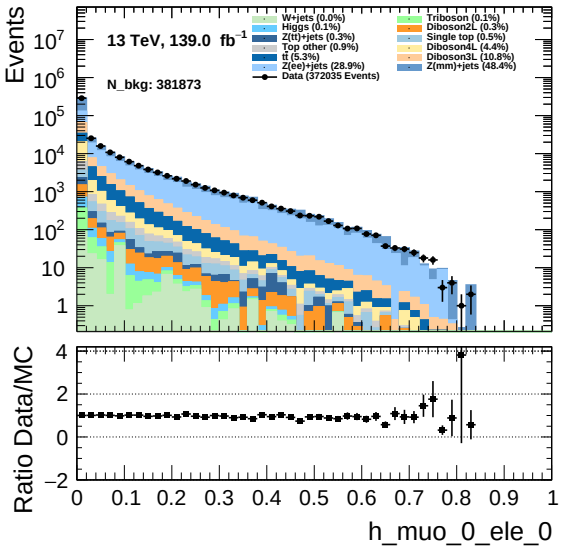


(d) Longitudal component for the second muon.

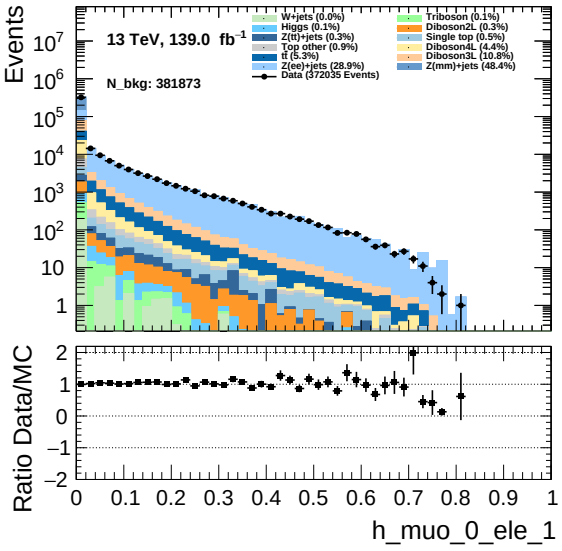
Figure 7: Longitudal component for individual particles, here jets and muons.



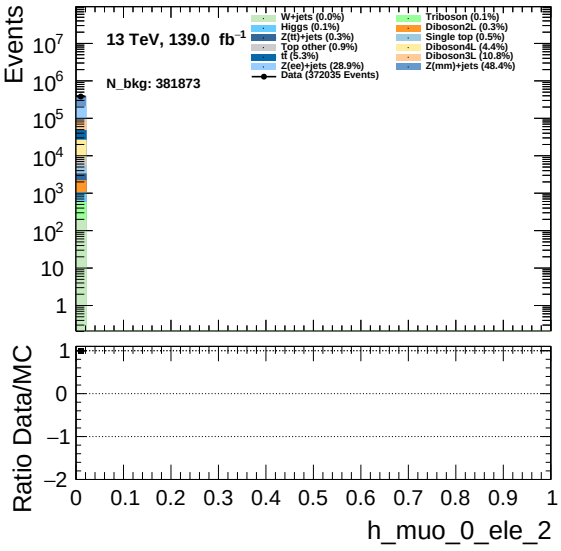
(a)



(b)

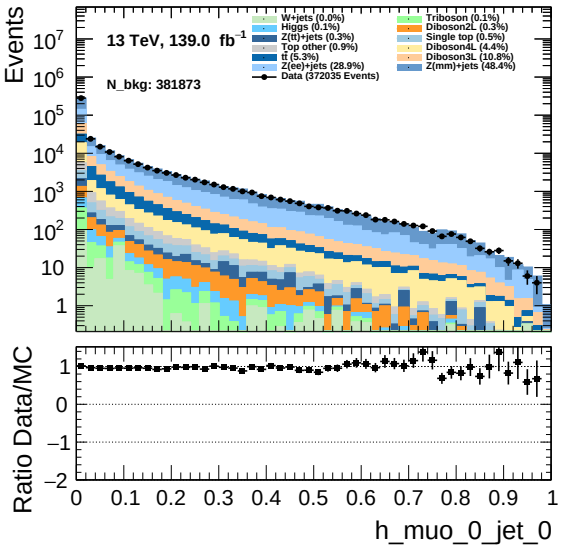


(c)

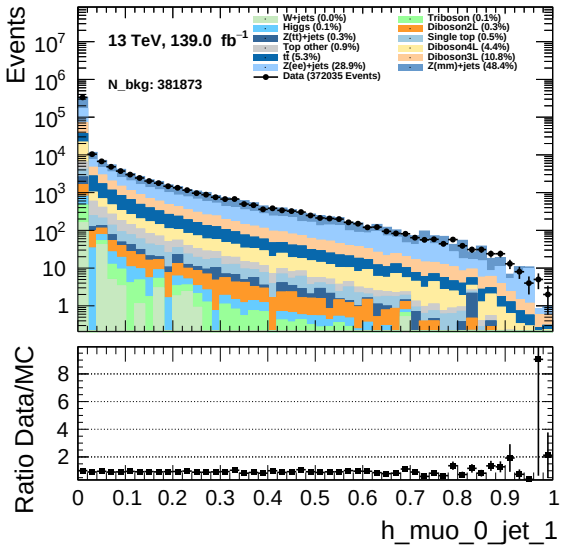


(d)

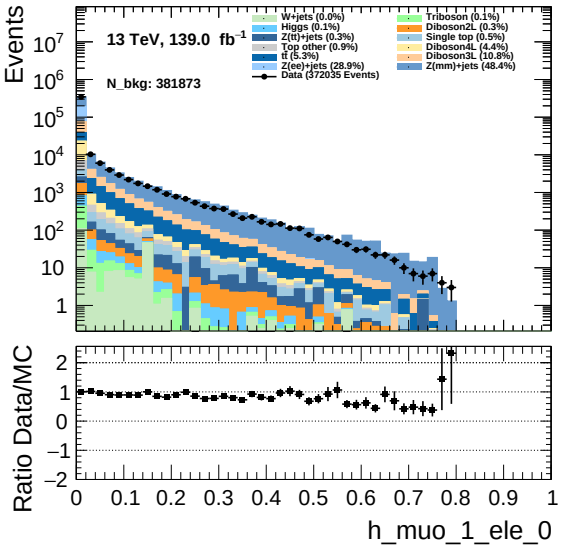
Figure 8



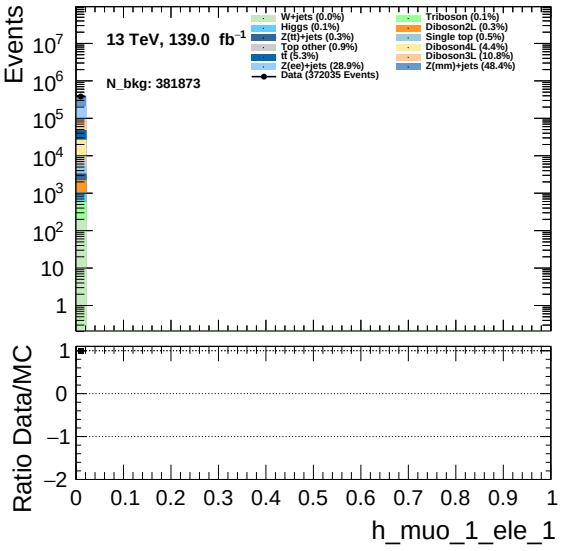
(a)



(b)

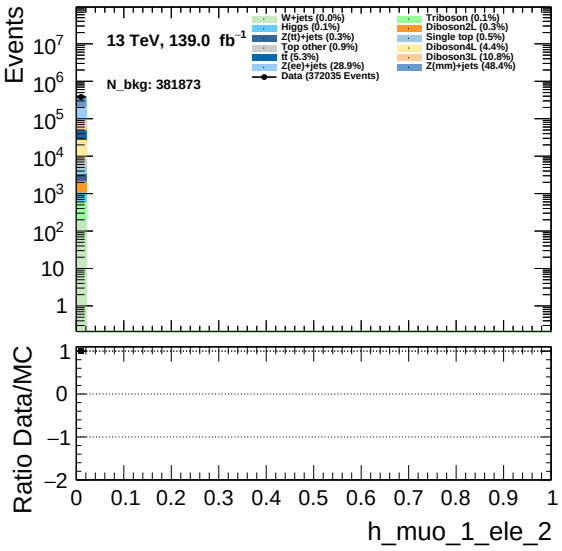


(c)

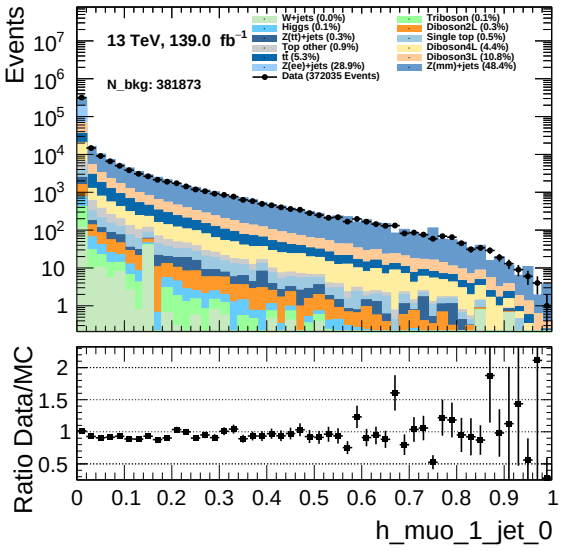


(d)

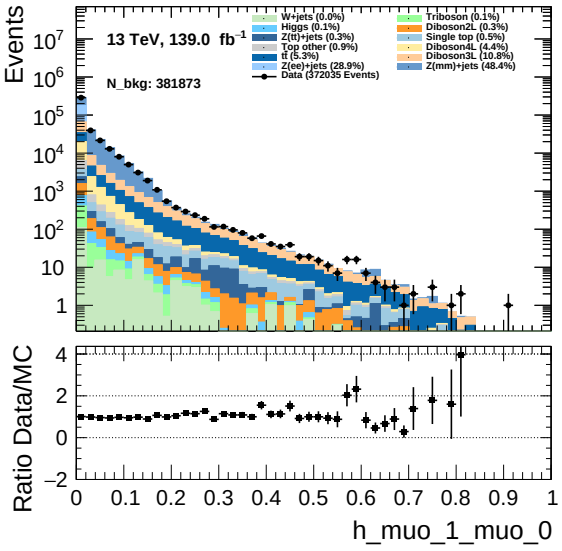
Figure 9



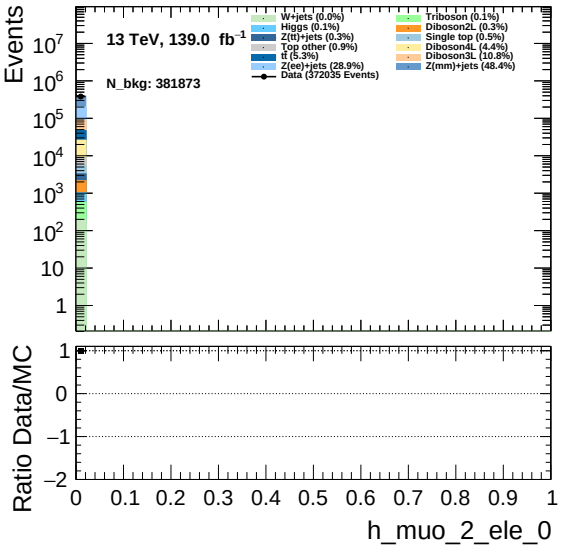
(a)



(b)

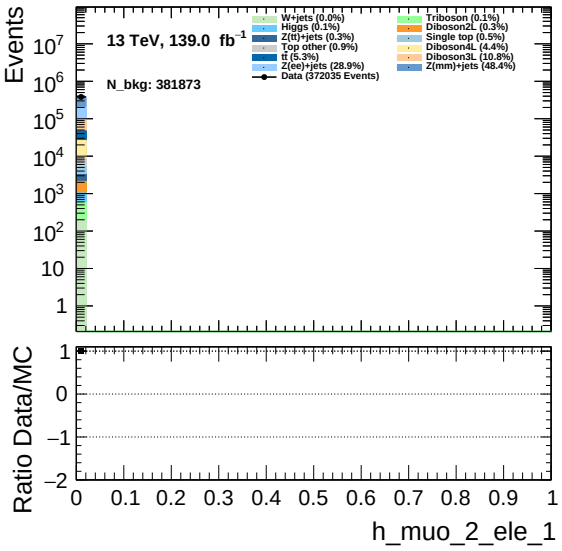


(c)

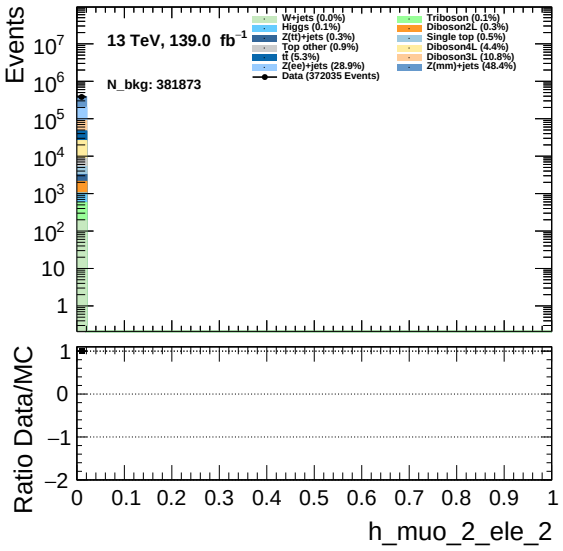


(d)

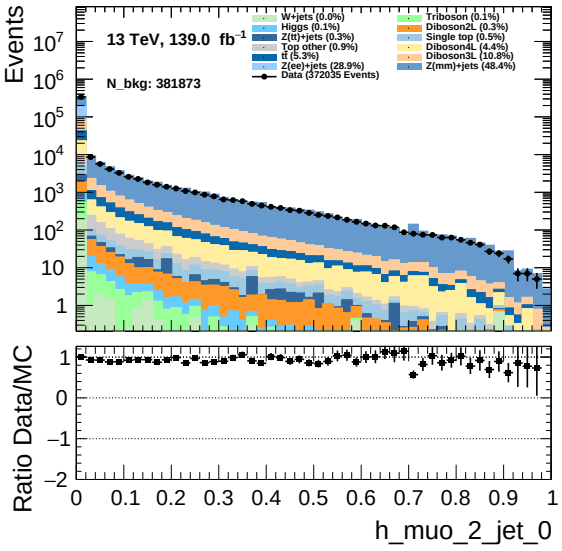
Figure 10



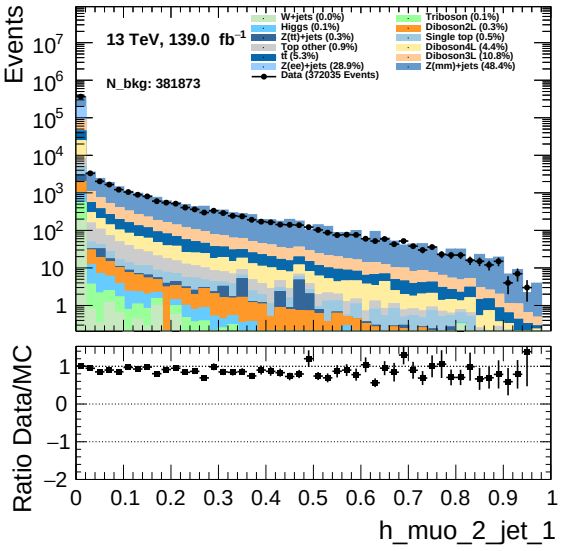
(a)



(b)

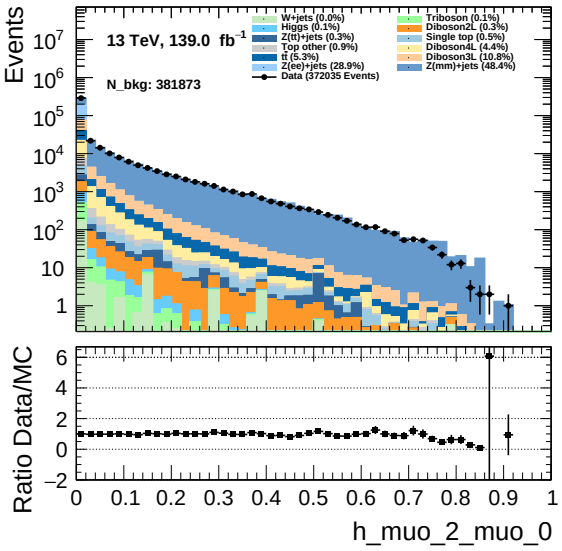


(c)

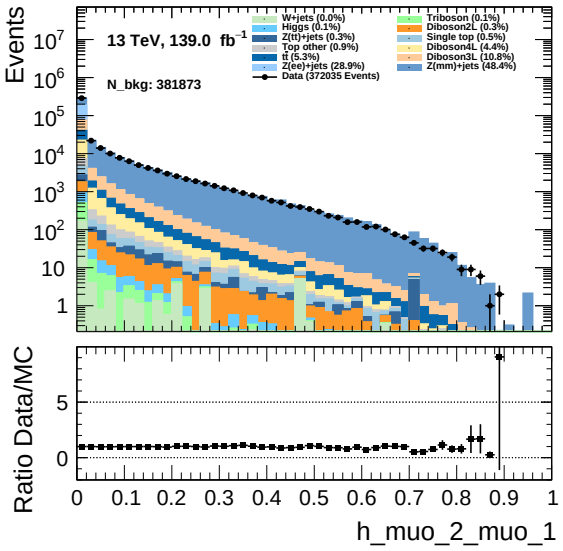


(d)

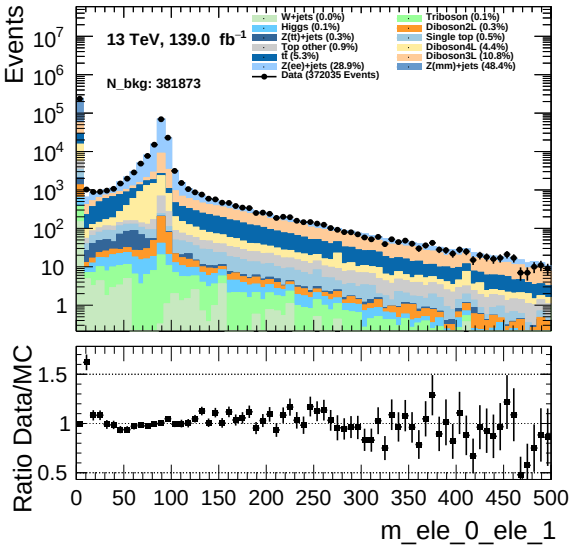
Figure 11



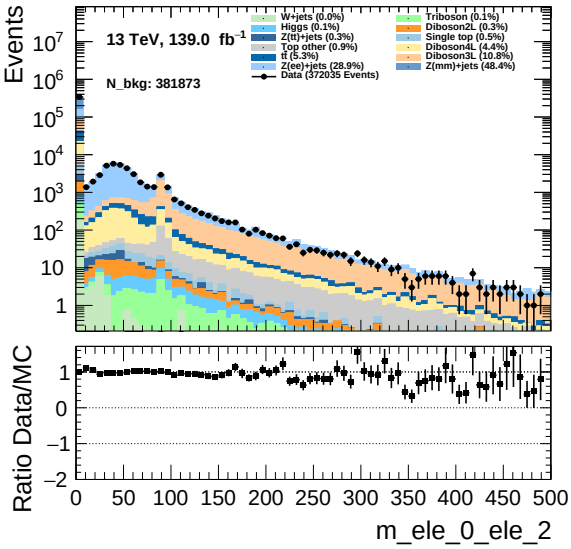
(a)



(b)

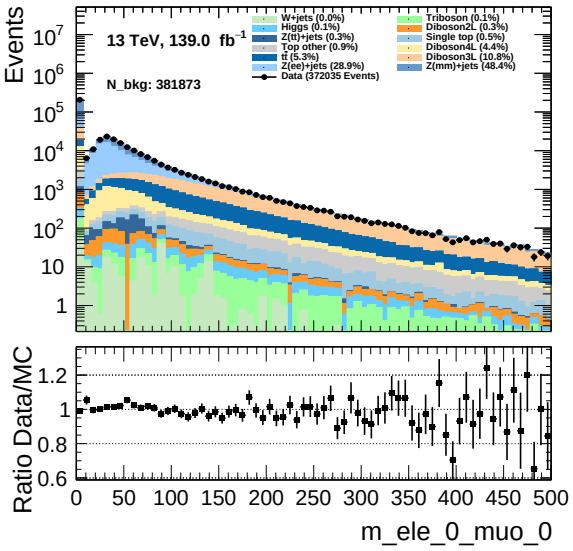


(c)

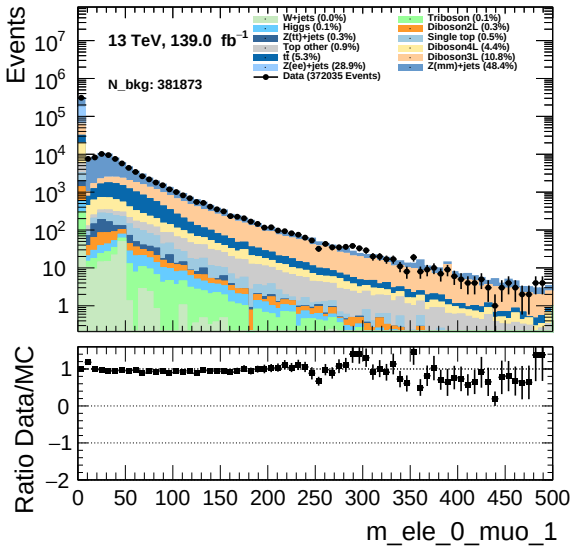


(d)

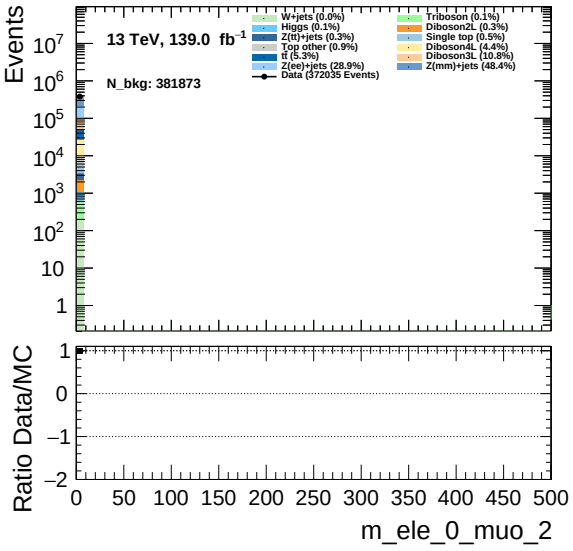
Figure 12



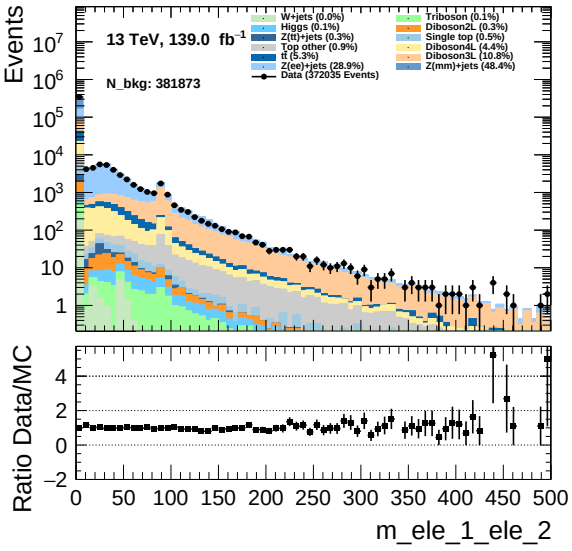
(a)



(b)

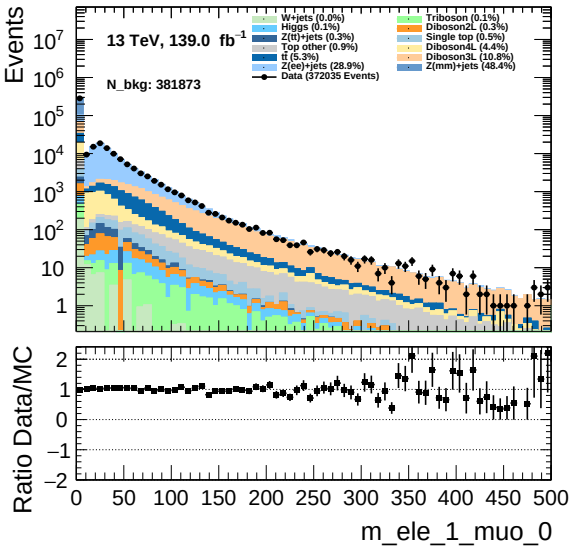


(c)

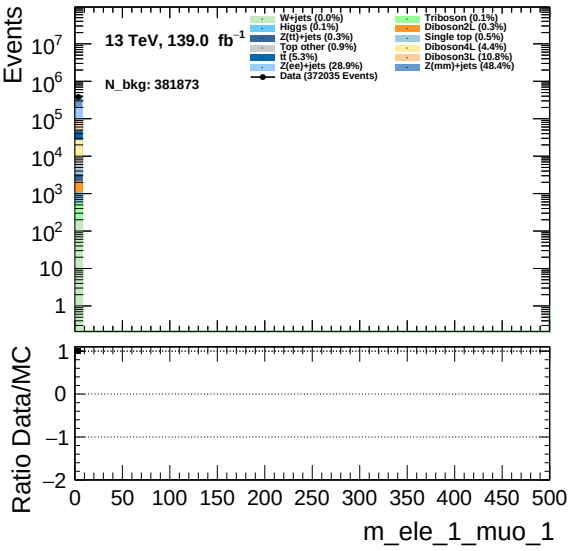


(d)

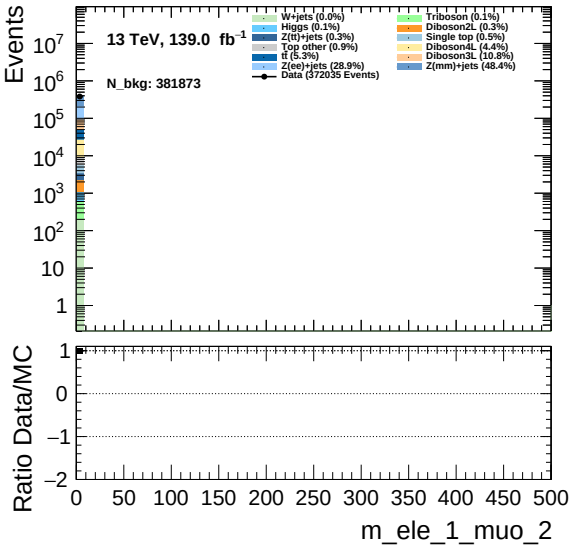
Figure 13



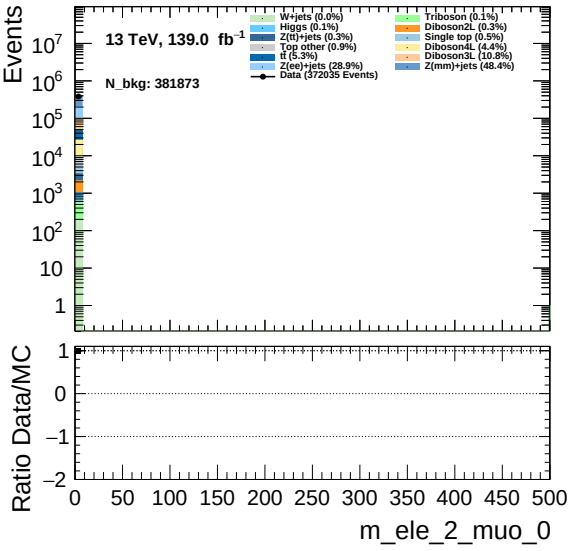
(a)



(b)

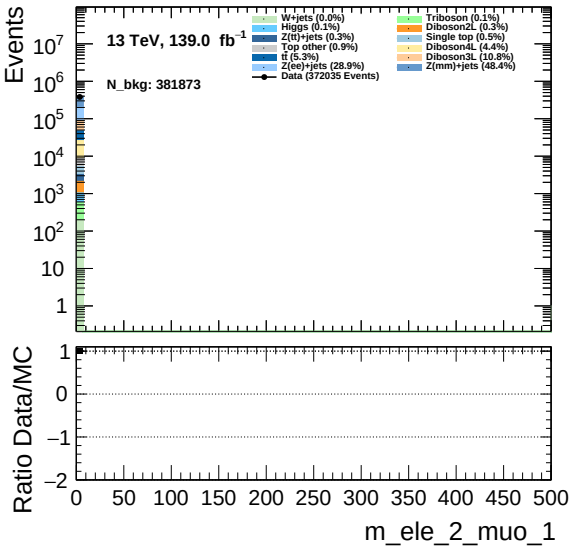


(c)

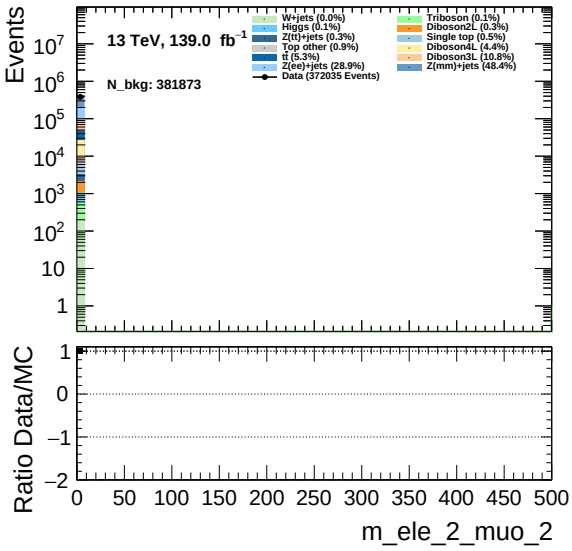


(d)

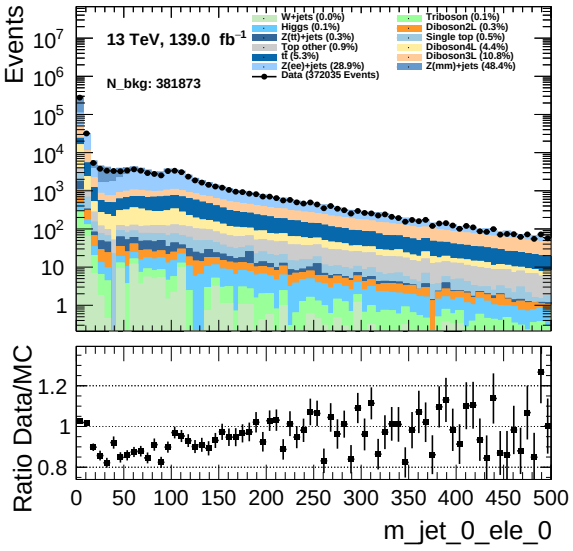
Figure 14



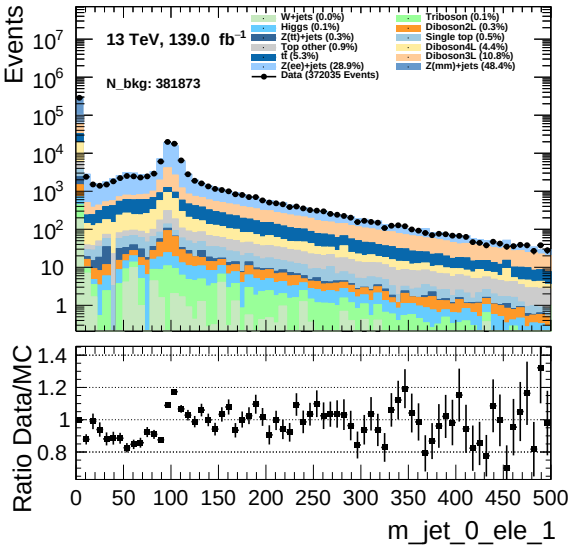
(a)



(b)

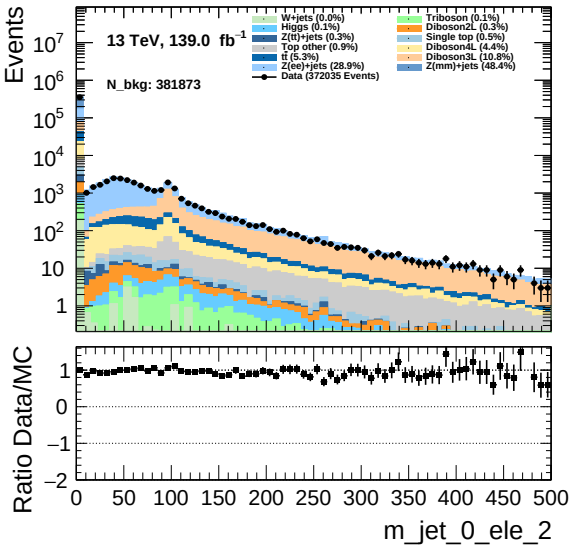


(c)

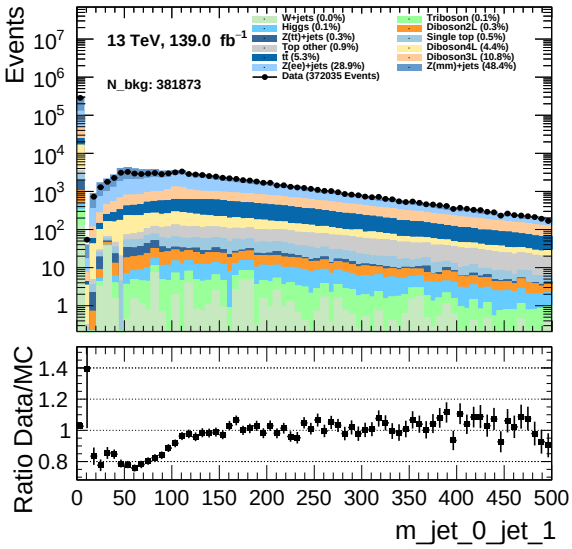


(d)

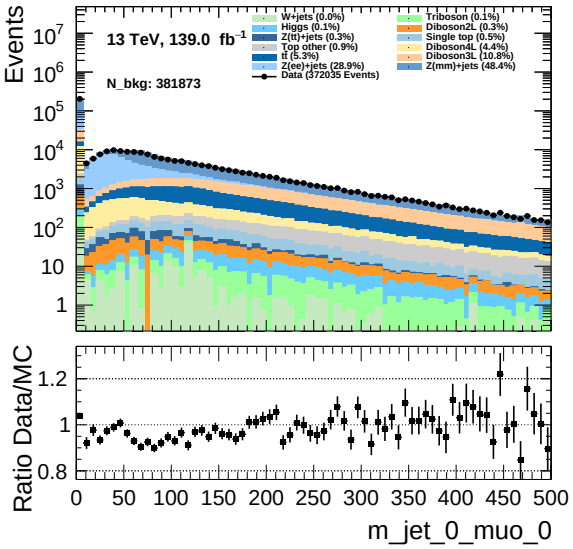
Figure 15



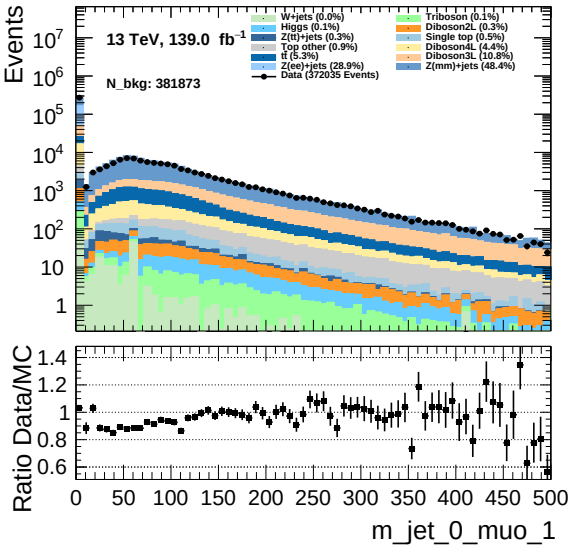
(a)



(b)

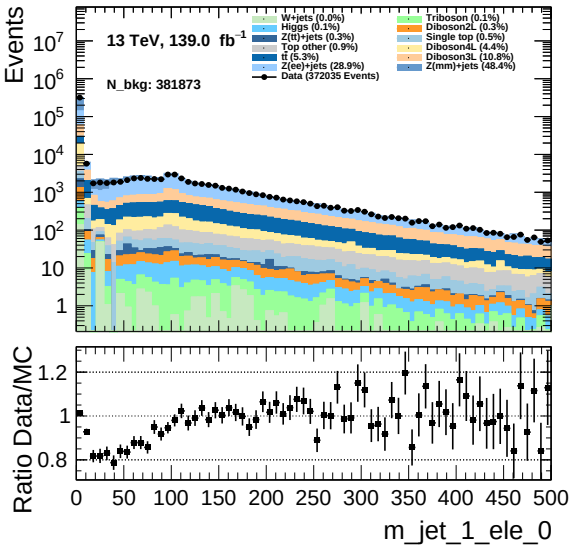


(c)

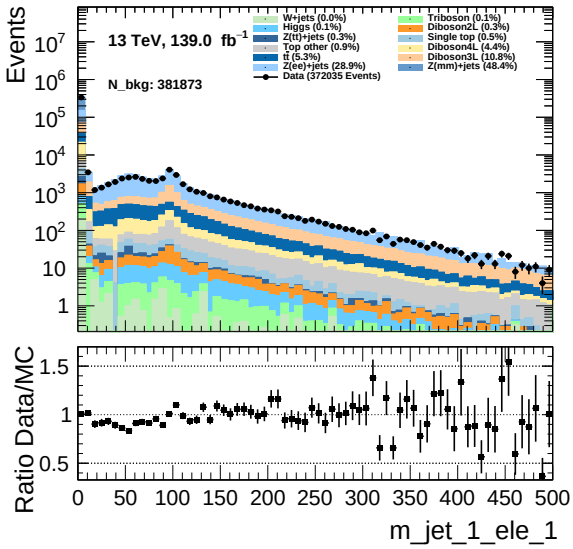


(d)

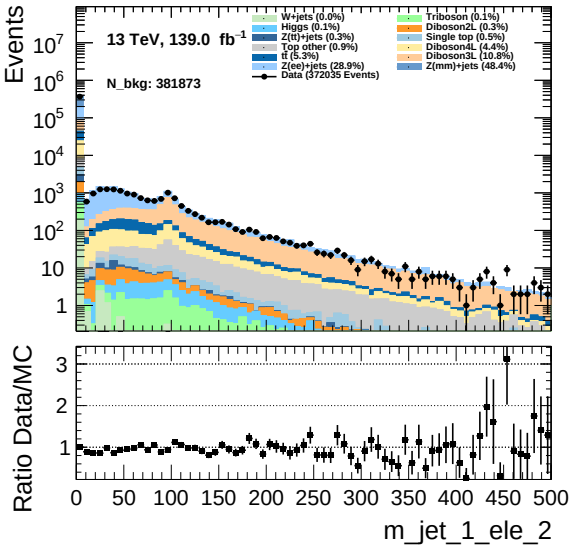
Figure 16



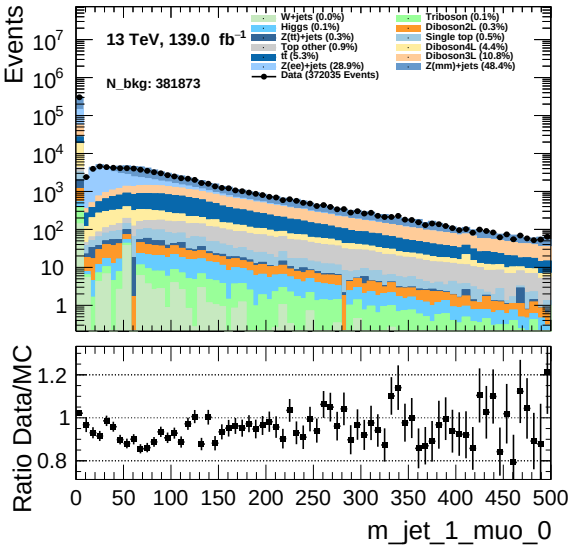
(a)



(b)

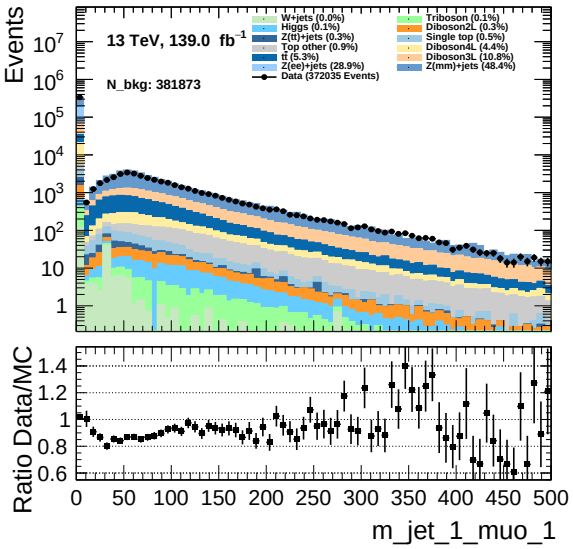


(c)

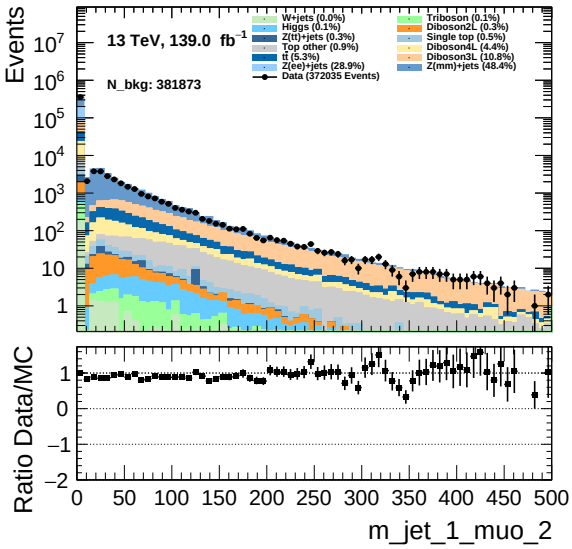


(d)

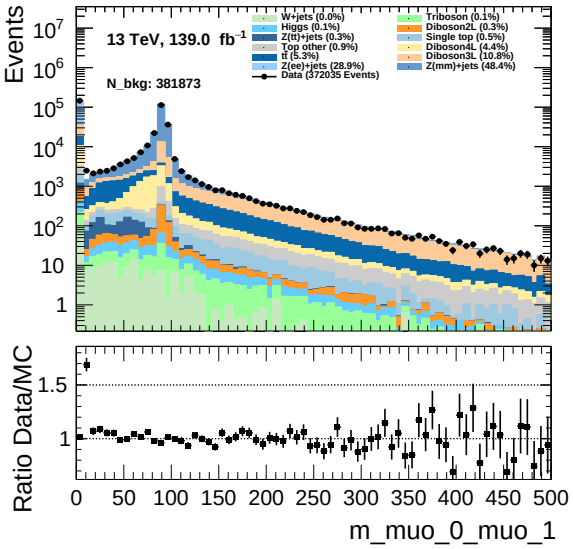
Figure 17



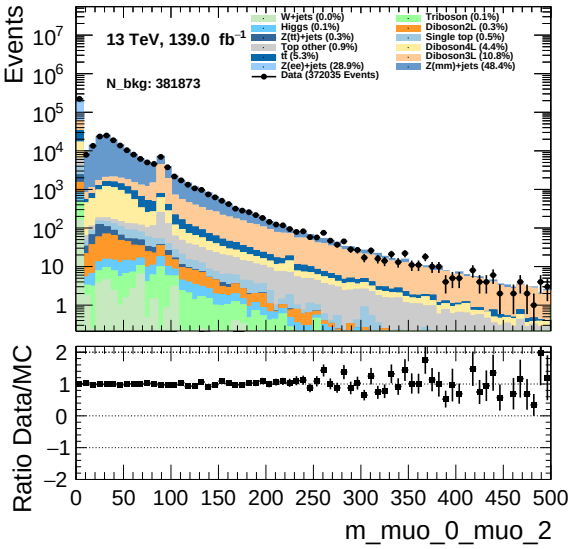
(a)



(b)

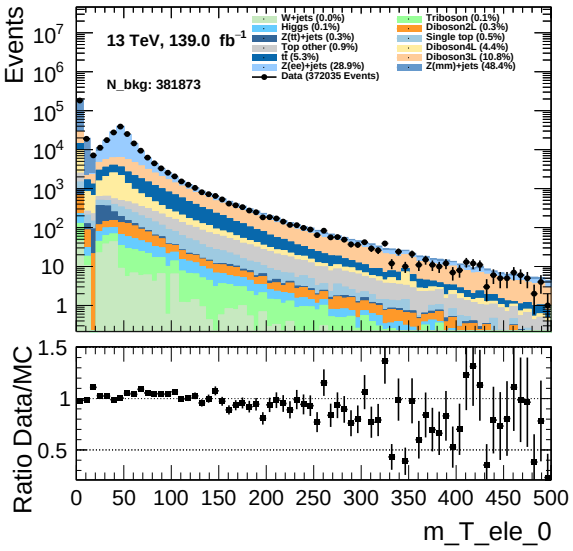


(c)

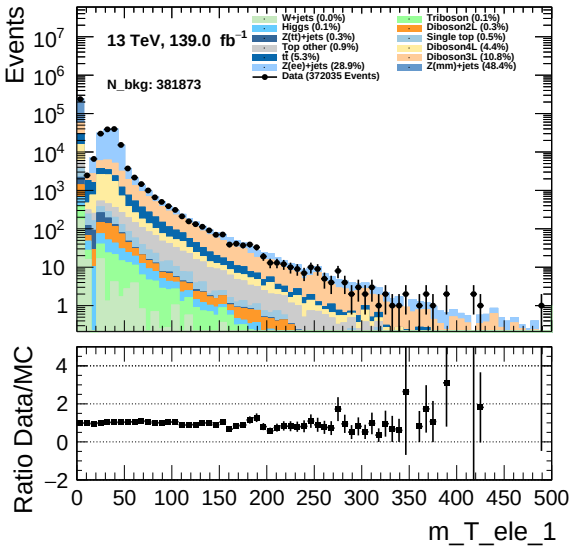


(d)

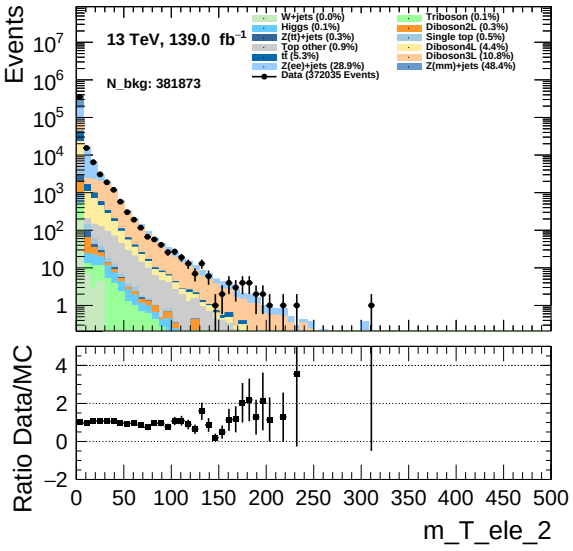
Figure 18



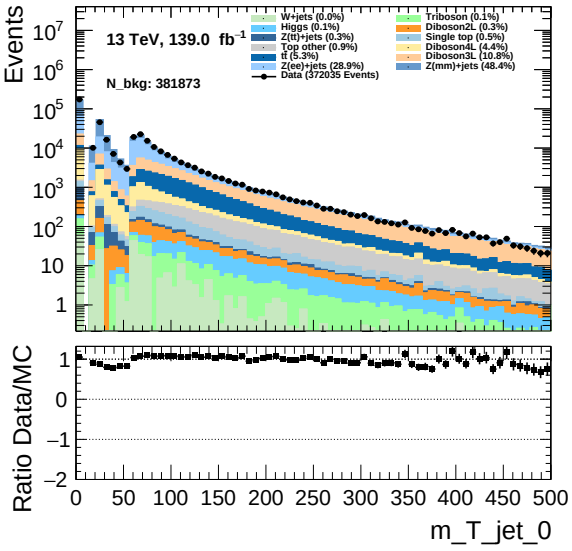
(a)



(b)

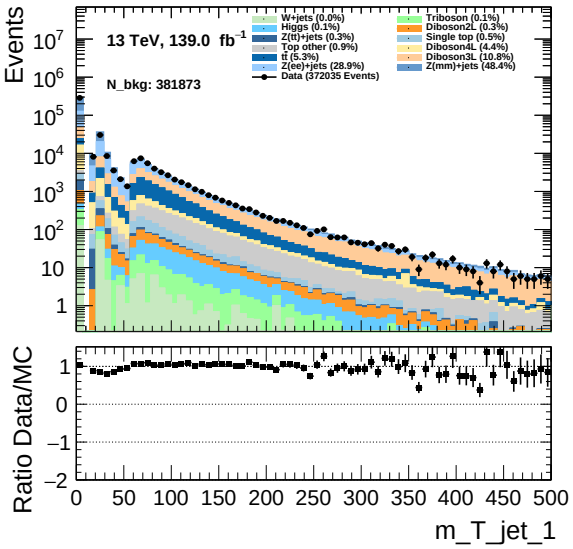


(c)

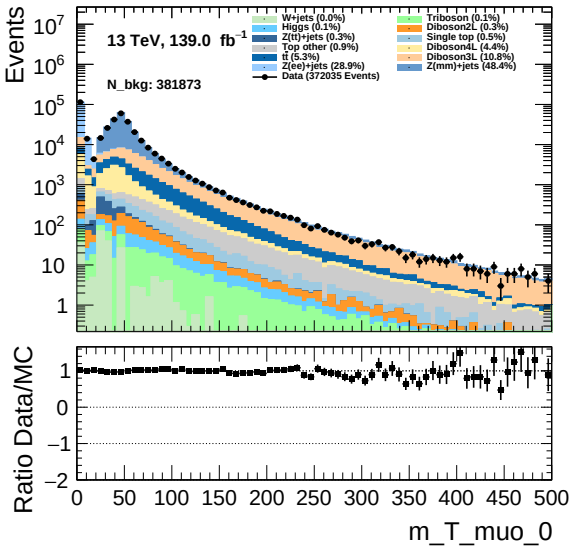


(d)

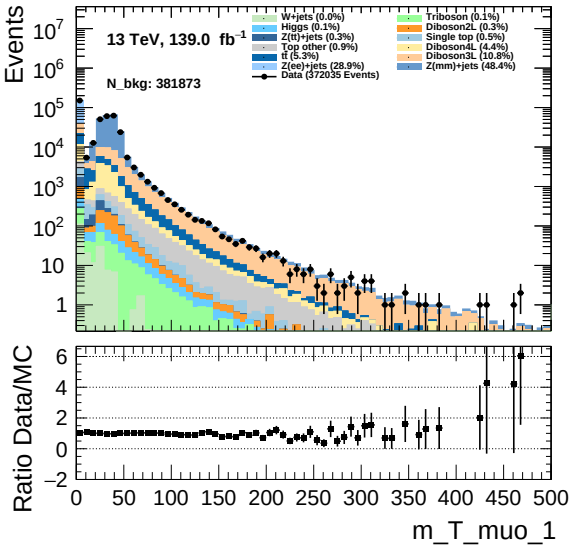
Figure 19



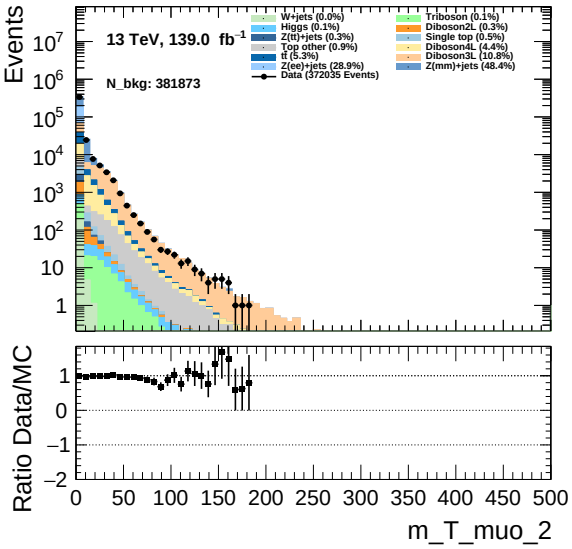
(a)



(b)

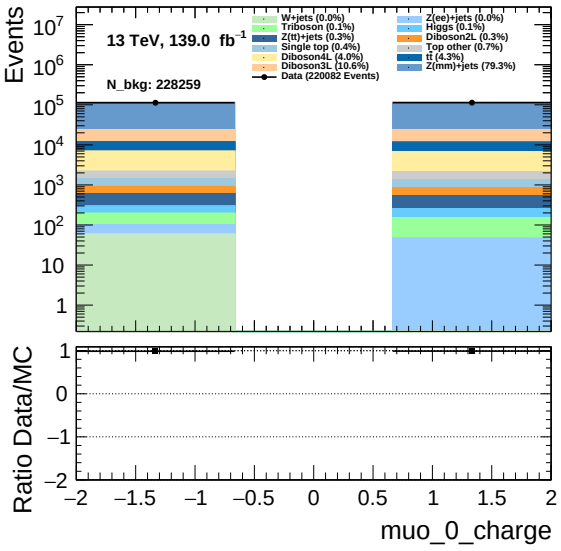


(c)

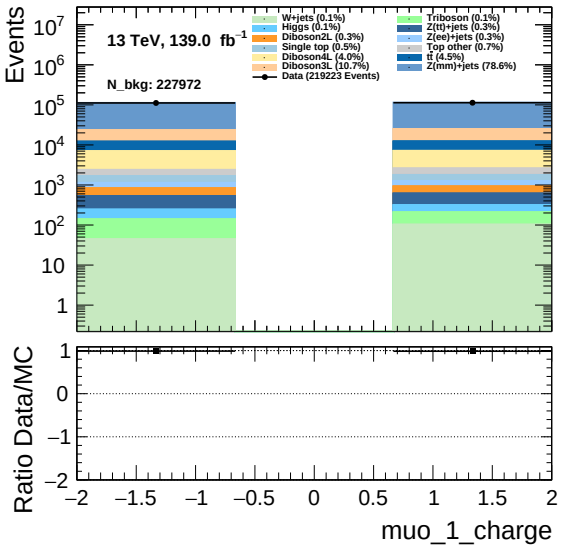


(d)

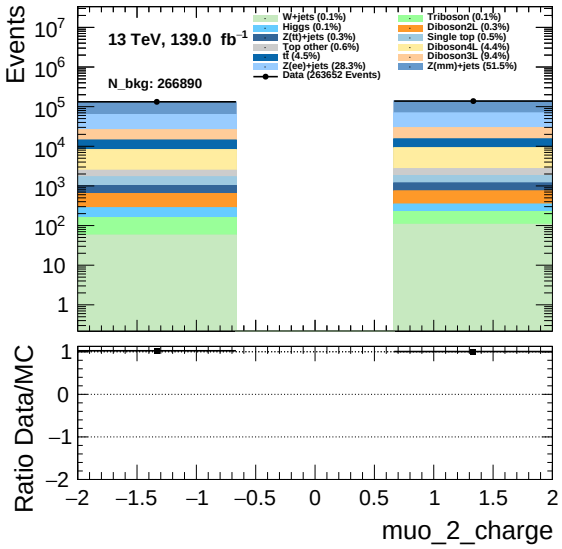
Figure 20



(a)



(b)



(c)

Figure 21

Appendix C

Appendix D

Bibliography

[1] V. Chandola, A. Banerjee and V. Kumar, *Anomaly detection: A survey*, *ACM Comput. Surv.* **41** (07, 2009) .

[2] S. Frette, W. Hirst and M. M. Jensen, *A computational analysis of a dense feed forward neural network for regression and classification type problems in comparison to regression methods*, .

[3] D. Rumelhart, G. Hinton and R. Williams, *Learning representations by back-propagating errors*, *Nature* (1986) .

[4] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*. MIT Press, 2016.

[5] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. 10.48550/ARXIV.1412.6980.

[6] The ATLAS collaboration, *Operation of the ATLAS trigger system in run 2*, *Journal of Instrumentation* **15** (oct, 2020) P10004–P10004.

[7] ATLAS collaboration, R. E. Owen, *The ATLAS Trigger System*, .

[8] S.-M. Wang, *ATLAS Computing and Data Preparation. ATLAS Induction Day + Software Tutorial*, .

[9] C. Bernius, *ATLAS Trigger and Data Acquisition. ATLAS Induction Day + Software Tutorial*, .

[10] R. Brun, F. Rademakers, P. Canal, A. Naumann, O. Couet, L. Moneta et al., *root-project/root: v6.18/02*, Aug., 2019. 10.5281/zenodo.3895860.

[11] E. Manca and E. Guiraud, *Using RDataFrame, ROOT's declarative analysis tool, in a CMS physics study. Using RDataFrame, ROOT's declarative analysis tool, in a CMS physics study*, .

[12] The HDF Group, *Hierarchical Data Format, version 5*, 1997-2022.

[13] S. Chekanov, *Imaging particle collision data for event classification using machine learning*, *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **931** (jul, 2019) 92–99.

[14] R. S. Geiger, D. Cope, J. Ip, M. Lotosh, A. Shah, J. Weng et al., *"garbage in, garbage out" revisited: What do machine learning application papers report about human-labeled training data?*, *CoRR* **abs/2107.02278** (2021) , [2107.02278].

[15] F. Chollet et al., *Keras*, 2015.

[16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro et al., *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015.

[17] T. O'Malley, E. Bursztein, J. Long, F. Chollet, H. Jin, L. Invernizzi et al., "Kerastuner." https://github.com/keras-team/keras-tuner, 2019.