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

by

Sakarias Garcia de Presno Frette

THESIS

for the degree of

MASTER OF SCIENCE

Faculty of Mathematics and Natural Sciences
University of Oslo

Autumn 2022

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

Abstract

Acknowledgments

- Veilledere
- William og Mikkel
- Foreldre
- NN

Contents

Introduction

Outline of the Thesis

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.

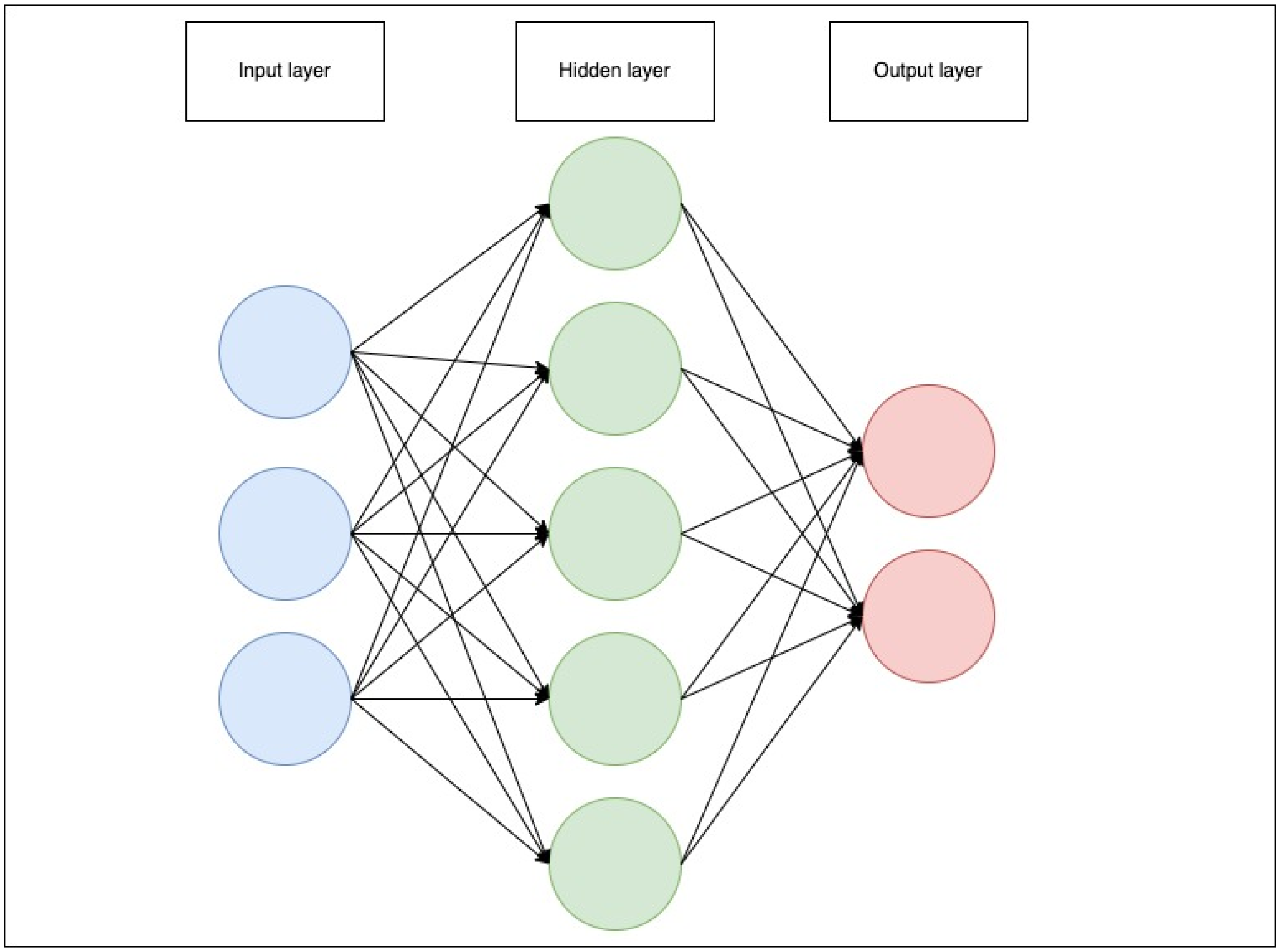

Figure 1.1: Simple neural network diagram drawm using Draw.io. Here the blue dots are the input layer, the green dots are a hidden layer, and the red dots are the output layer. The arrows shows the connections between each node.

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 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^l_{ij}	The weight connecting node i in layer $l-1$ to node k in layer l.	$\mathbb{R}$
b^l_j	Bias acting on node j in layer l.	$\mathbb{R}$
z^l_j	Node output before activation on node j on layer l.	
a^l_j	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:

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

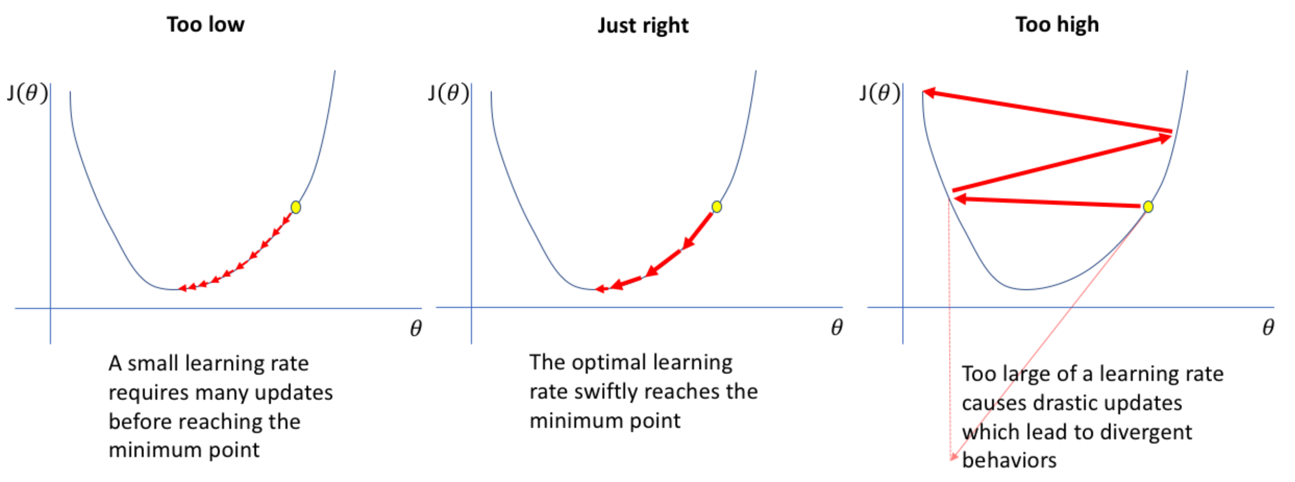

Figure 1.2: Figures showing different choice of learning rate for a given costfunction, with respect to the tunable parameters. Source: Jeremy Jordan, accessed 03.10.22.

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, 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 $a = \sigma(z)$.

[1]Give reference to hyperparameters

- 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.

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:

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

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.

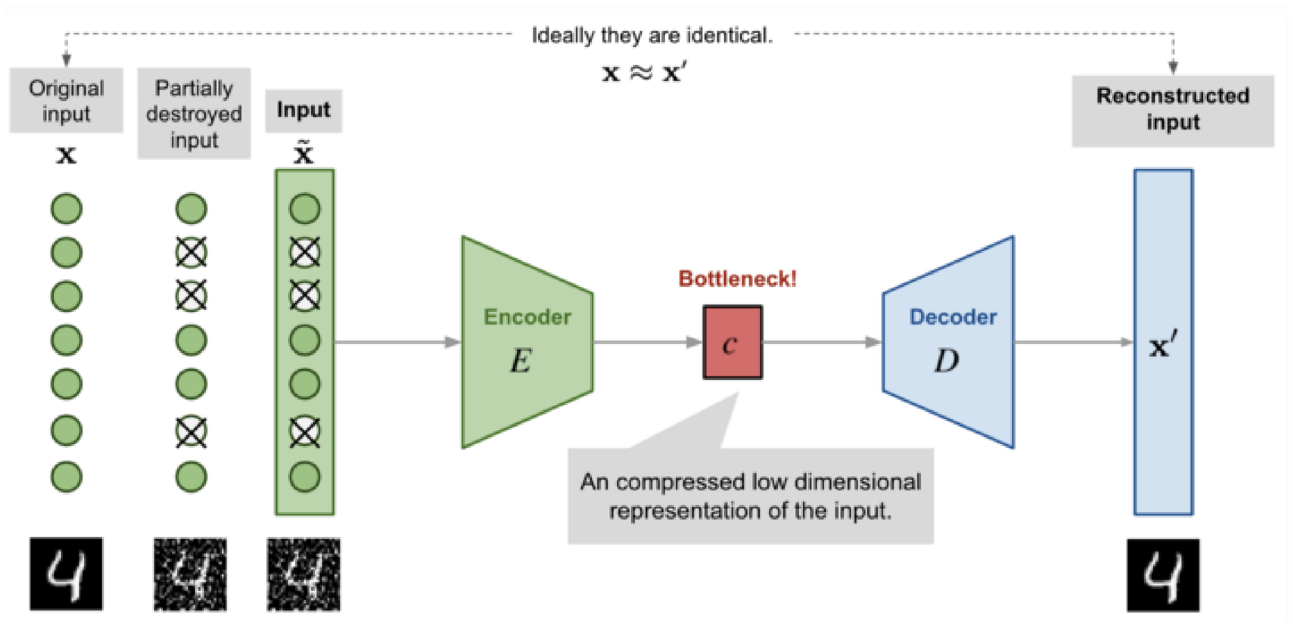
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Original input
$\mathbf{x}$
Partially destroyed input
Input
$\tilde{\mathbf{x}}$
Encoder
E
Bottleneck!
c
Decoder
D
Reconstructed input
$\mathbf{x}'$
An compressed low dimensional representation of the input.

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.

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

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

Chapter 2

Standard model phenomenology

Structure and composition of the Standard Model

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

Chapter 3

Implementation

ROOT

It is a lot. [5]

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

RDataFrame is a

In the codelisting we see an example of a C++ function used by RDataFrame.

```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;
}
```

In the codelisting below we see a simple implementation of RDataFrame in python.

	$jetP_T$	$jetPhi$	$lepP_T$	$lepPhi$
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

```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")

    # 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",
    )

```

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 [6].

!! Motivation for using such a matrix in machine learning → 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. 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

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

Code implementation

Chapter 4

Results

Chapter 5

Discussion

Conclusion

Future work, more work

Appendices

Appendix A

Appendix B

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] 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.

[6] 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.