Searching for new Physics at the LHC with Machine Learning

by

Lavronenko, Kostiantyn

Bachelor's Thesis

Supervised by
Prof. Dr. Michael Krämer
Institute for Theoretical Particle Physics and Cosmology
RWTH Aachen University

Aachen
September 13, 2021

Contents

Page

1 Introduction **2**

2 Jet production and data simulation **4**

3 Deep Learning **7**

3.1 Deep Learning Basics 7

3.2 Convolutional Layers and Poolings 9

3.3 Edge Convolutoinal Layer 10

3.4 Performance Measurement 12

4 CNN classifier **14**

4.1 Jets as images 14

4.2 Architecture 14

4.3 Results 15

5 LundNet classifier **17**

5.1 Data-preparation 17

5.2 LundNet architecture 20

5.3 NeighboursNet architecture 22

5.4 Results 23

6 Discussion and Conclusion **25**

A Appendix **27**

A.1 CNN architecture from chapter 4 27

A.2 CNN results for 300k signal and background images 27

A.3 Jet declustering from section 5.1.1 28

A.4 Lund plane construction from section 5.1.2 29

A.5 LundNetTF architecture from section 5.2 30

References **31**

1 Introduction

The Standard Model (SM) describes three of the four fundamental physics interactions: the electromagnetic, the weak and the strong force. Experimental discoveries have already confirmed a lot of significant predictions, that Standard Model postulated. One of them is the last fundamental particle, the Higgs Boson, discovered at Large Hadron Collider in 2012. Though there have been a lot of affiramtions of Standard Model behaviour, the LHC continues to look for physics Beyond the Standard Model (BSM), as the Standard Model is incomplete and needs adjustments. Known issue with the Standard Model are inability to explain the nature of dark matter and matter-antimatter asymmetry. That is why the search for anomalies, called signals in physics, in a big amount of data, produced by the LHC, is a significant challenge, that would contribute to new physics.

After a collision of two protons at the LHC new particles are created. Those particles are decaying on the way to detector into other. If a BSM particle is created after a collision it splits into SM particles. Particles carrying a colour charge, such as quarks, cannot exist in free form because of QCD confinement, which only allows for colorless states. When a particle decays its' fragments carry away a colour charge. To obey confinement coloured fragments create new coloured objects to be in colourless state and propagates in a narrow cone to the detector. This narrow cone is called a jet. If a heavy BSM particle is boosted enough all subjets from decay products might end up in one (fat) jet. One of the main tasks is to separate high-energetic signals from background. In this work top jets will be referred as signals and quantum chromodynamics jets (QCD) as background. QCD jets are jets from gluons or up, down, strange or charm quarks.

One way to search for signals in a big amount of data is using Machine Learning. Machine Learning methods have already proven their significance in a scientific world and their innovative models and algorithms allow physics to expand horizons in searching for the BSM [1].

Jets consist of multiple particles, that were collected in the detector. To work with Machine Learning algorithms simulated jets are used, because actual events, produced in the real world, contain a lot of yet inexplicable behaviour. Those jets are simulated by parton shower Monte Carlo event generators [2], [3].

One of the most notable Machine Learning branches is Deep Learning. The way Deep Learning is used in this work is to create trainable neural networks, that can differentiate between multiple inputs. This is called a classification task, which is a part of supervised learning. Out of the different types of networks Convolutional Neural Networks (CNN) have shown remarkable results in the Deep Learning community on classifying different pictures [4], which will be used to classify top and QCD jets.

Even though the CNN is a revolutionary method for a tagging task, one may ask if it is possible to introduce more physics into neural networks. One way to do so is to create a LundNet, described in references [5] and [6]. Using properties of generated particles we can apply a reclustering algorithm [7] and create a particle tree, which can serve as an input to a neural network. This neural network would then be able to create trainable weights, applying to edges [8] of each node of a reclustered tree, which would then create a constriction to classify a type of jet, which was fed into it.

This work is structured as followes: In chapter 2 theory of the jet physics is discussed. Chapter 3 describes tools of Machine and Deep Learning, that are used in this work, for better a understanding of how neural networks are build. In chapter 4, the CNN architecture is presented with respective results. Chapter 5 explains the reclustering algorithm, needed for the pre-processing of the data, simulated as described in reference [9], and how reclustered particles are presented in a Lund Plane. Moreover a neural network (LundNet) architecture is presented with its' results on a pre-processed Lund Plane. Furthermore, two more architectures, modified from reference [10] are applied to a Lund Plane and presented with it's results together with LundNet implementation in the chapter 5. In chapter 6, results of all models are discussed in comparison to results of the LundNet in reference [6], ParticlNet in reference [10] and Thorbens Finke Master's Thesis [9] at RWTH Aachen.

2 Jet production and data simulation

Top qaurk is the heaviest elementary particle. Top quark lives in the center of many new physics models motivated by the hierarchy problem [11]. As Standard Model is not complete many shortcomings and extensions to it are used, like supersymmetry or little Higgs models, that predict top partners which naturally decay to top quarks. Some other models, like extra dimensions with geometries linked to the mass hierarchy in the Standard Model, predict large couplings of new states to top quarks, which again induce the interest in top quarks that can influence new physics.

That is why as a benchmark signal we use $t\bar{t}$-quarks. The top quark decays via the weak force with a mean lifetime of $5 \cdot 10^{25}$ seconds. Top quarks lifetime is a lot shorter than the time at which strong forces of QCD act and it decays before hadronisation. That is why a top quark can be considered as a "free" particle.

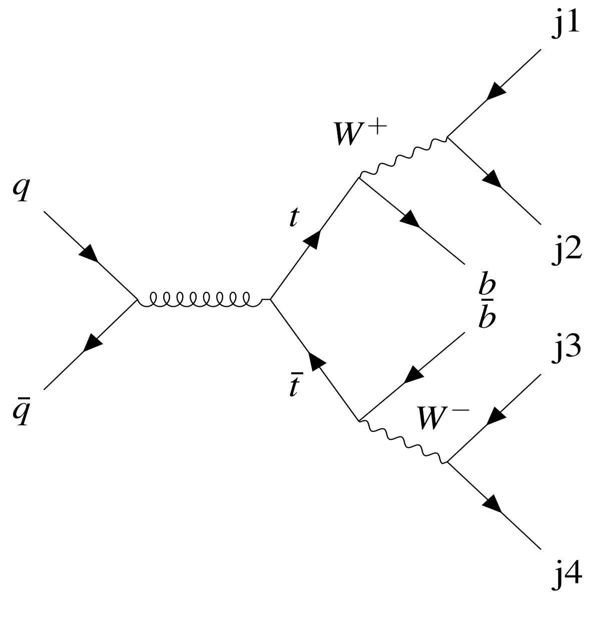


Figure 2.1: Feynman Diagram for the production of a hadronicaly decaying $t\bar{t}$-Pair. Diagram from [9]

After a top quark is created, as shown in a Feynman diagram in Figure 2.1, they decay into a W-bosons and a bottom quark. The W-boson further can decay hadronicaly (in 68%) in two jets or leptonicaly (in 32%) in one lepton and neutrino. In this work we use hadronicaly decaying $t\bar{t}$-quarks, that is why W-boson decays via the weak interaction into two other quarks. Quarks can not exist without having colour partners, they need to be in a colorless state, which implies that they need to create new colour particles around them and propagate to a detector as a fat jet. This creates 3 jets for each top quark. If a top jet is boosted enough, those three jets will be captured at the detector close enough to each other, to determine a top jet.

Background data consists of light QCD jets. It is produced by collision of two protons that further produce di-jet of (anti-)up, (anti-)down, (anti-)charm, (anti-)strange quark or a gluon. Each proton beam has an energy of 7 TeV. Created di-jet firstly consists of two fat jets. Initial state radiation can lead to additional jets. Nevertheless, all those jets are of a light QCD nature, that is why they are used as background.

Data, used for training and testing the models, is simulated, using different event simulation tools [2], [3]. Events are generated on parton level using MadGraph5 aMC@NLO 2.6.5 [12]. With PYTHIA 8.2 [13] the parton shower and hadronization are simulated. Delphes 3.4.2 [14] is used for a detector simulation. Clustering of jets is performed with the anti-k_t algorithm implemented in FastJet [15]. Simulation of the data and tools are described in more details in the reference [9].

The output of the simulation chain are events with particles 4-momenta. In the chapter 5 we use 4-momenta of those particles to transform the data into the Lund Planes, which afterwards are used as an input into a neural network. Those events could as well be represented as callorimeter pictures. In the figure 2.2 both, top (on the right) and QCD (on the left) jets, are shown in 40x40 pixel pictures. Unfortunately, a task to differ between two pictures is not always easy for a naked eye. That is why in section 4 those pictures are taken as an input into the CNN model to classify them.

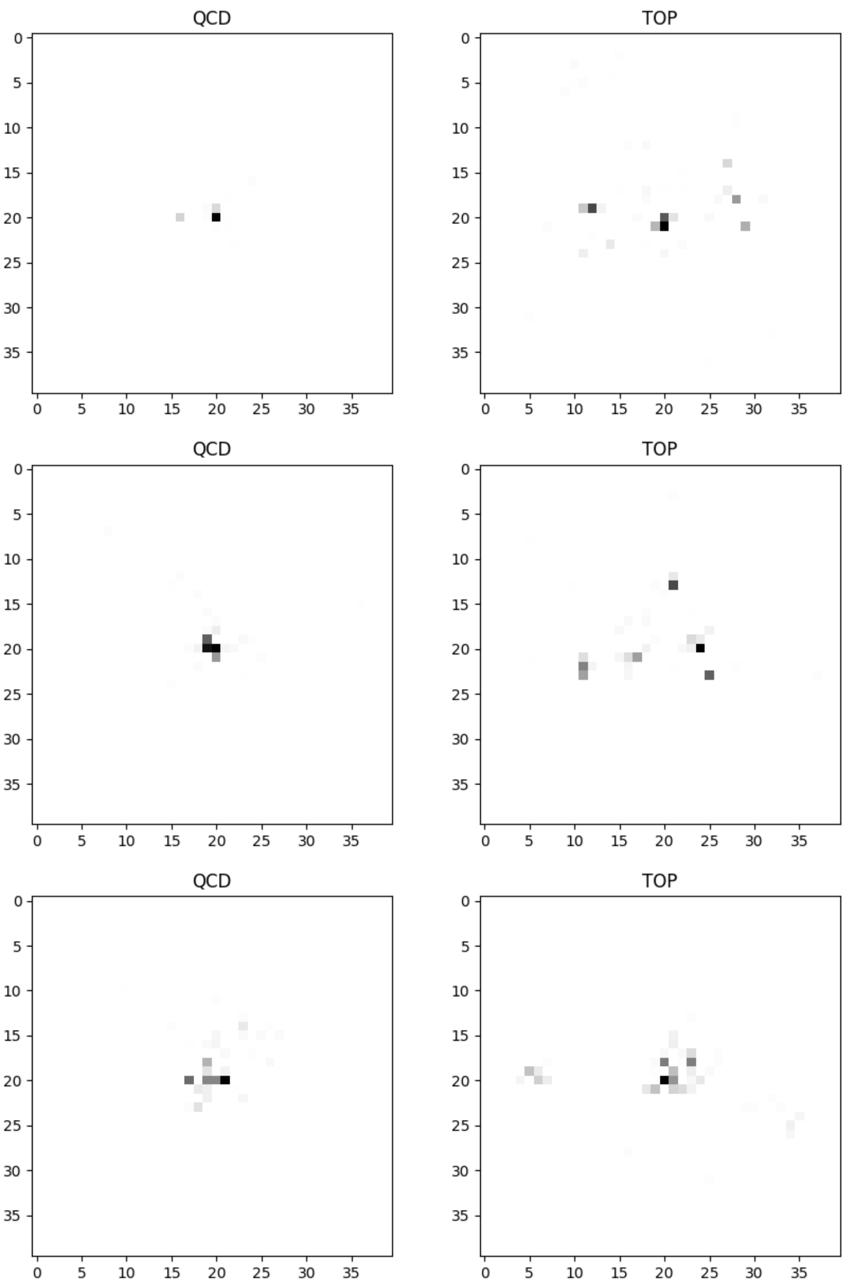


Figure 2.2: Left: images of captured QCD jets. Right: images of top jets. It can be seen that some of top jets show the behaviour described in chapter 2, they decay into 3 other jets. But this is not always the case, picture in the bottom right corner is a good example for one fat jet, where 3 emissions of the top jet are indistinguishable.

3 Deep Learning

In this chapter Deep Learning and its tools are introduced. The basics of deep learning are taken from following course [16]. It is also recommented to look at Ref. [17] for a better and deeper understanding of deep learning concepts.

3.1 Deep Learning Basics

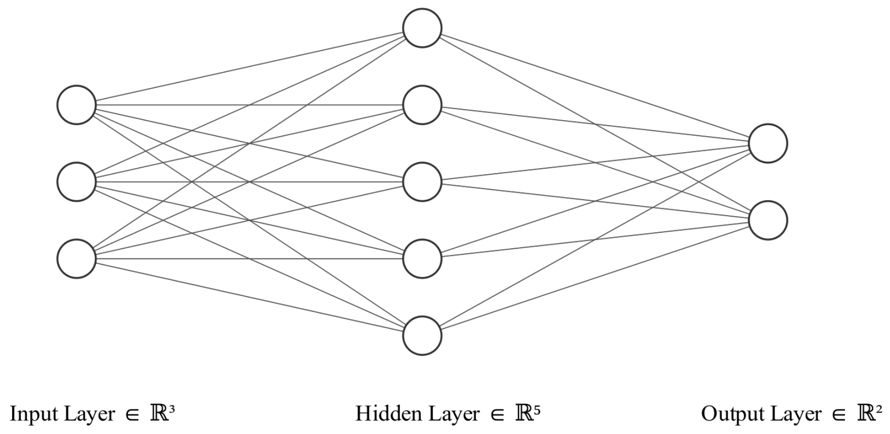

Input Layer $\in \mathbb{R}^3$
Hidden Layer $\in \mathbb{R}^5$
Output Layer $\in \mathbb{R}^2$

Figure 3.1: Simple Example of a Dense Neural Network (DNN)

Deep Learning idea is to create artificial neural networks, that, via a set of mathematical operations, can extract specific features from the input data and learn to perform better for a given task. Learn means to improve a set of parameters used in those mathematical operations. The process of learning is as well called training and the parameters are correspondingly called trainable parameters. Some examples of tasks, suitable for deep learning, are classification, like picture recognition, or regression.

Classification task is the main emphasis of this work, that is why we use a dense neural network (DNN) from the figure 3.1 as an example. The task is to predict the possibility of the input data belonging to some class ($y_{predicted}$) as closely as possible to the true class of the input (y_{true}). Each neural network consists of input, hidden and output layers. Every circle is referred as a node, or neuron, of a neural network and DNN is a fully connected neural network, which means that every every node is connected to all nodes of the neighbouring layers. Each layer is an output of mathematical operations between trainable parameters and nodes of the previous layer, except for the input layer. In our example, input layer is the data, that needs to be classified, and output layer corresponds to the classes, to which input data needs to be classified (in Fig. 3.1 two nodes correspond to two classes). Layers between input and output layers are called hidden layers.

Each node of the hidden layer in the figure 3.1 performs calculation

$$n(\mathbf{x}, \mathbf{w_1}, \mathbf{b}_1) = \mathbf{w_1} \cdot \mathbf{x} + \mathbf{b}_1 \tag{3.1}$$

with $\mathbf{w}_1$ and $\mathbf{b}_1$, called weights and biases respectively, trainable parameters, and $\mathbf{x}$ the data, nodes, from the input layer. The output layer then calculates its nodes

$$y(\mathbf{n}, \mathbf{w_2}, \mathbf{b_2}) = \mathbf{w_2} \cdot \mathbf{n} + \mathbf{b_2} \tag{3.2}$$

with $\mathbf{n}$ corresponding to nodes of the previous hidden layer and $\mathbf{w}_2$ and $\mathbf{b}_2$ a second set of trainable weights and biases. We unite all the trainable parameters in one symbol $\theta = (w_1, b_1, w_2, b_2)$ and name the output layer as our prediction and obtain the equation

$$y_{prdicted} = y(\mathbf{x}, \Theta). \tag{3.3}$$

For more complex neural networks with multiple-layer-representation a simple matrix multiplication between weights and nodes would squeeze into one. That is why activation functions are needed. Instead of just multiplying weights with inputs and adding biases to it, we also apply non linear activations σ

$$n(\mathbf{x}, \Theta) = \sigma(\mathbf{w} \cdot \mathbf{x} + b) \tag{3.4}$$

to it. There are several activation function used in the literature, but most commonly used are rectified linear unit (ReLu)

$$ReLu(x) = max(0, x) \tag{3.5}$$

and softmax

$$softmax(x_i) = \frac{exp(x_i)}{\sum exp(x_j)}. \tag{3.6}$$

ReLu is used to avoid linearities and merging all weights together and softmax is mostly used for the probability calculation before the output layer. Softmax is used as probability because prior components of some node x might be negative, bigger than one ore not summed to one, but after applying softmax from equation 3.6, every node x_i will be in the interval [0,1] and sum of all nodes would give one, corresponding to 100%.

To update, train, the parameters it is essential to define the objective function which describes how outputs differ from the real classes of the input. In this work we use the categorical cross entropy

$$J(\Theta) = CCE = -\sum_{i=1}^{n_c} y_{true,i} \cdot ln(y_{predicted,i}), \tag{3.7}$$

as it is mostly suitable for the classification task. The reason why CCE is used for the most classification tasks is because for the highest matching between y_{true} and $y_{predicted}$ CCE is minimized.

After defining the objective (loss) function the gradient $\frac{dJ}{d\Theta}$ has to be calculated and subtracted from the weights: $\Theta \to \Theta - \alpha\frac{dJ}{d\Theta}$. This step is called learning. The hyperparameter α is a learning rate, which defines the step size at each iteration while moving toward a minimum of a loss function. This procedure can be repeated until the objective has achieved its minimum.

To train a model, several fundamental hyperparameters, besides the learning rate, have to be chosen. It is normally not practical to update the parameters of the whole training set at once. Batch size defines the amount of data that is propagated through the network, so that a system would not be overloaded with huge amount of data at once. This interval method for optimizing the objective function is called a stochastic gradient descent. Adam optimizer [18] is based on a SGD to update learning rate after every epoch, i.e. every run through the whole data. Number of epochs defines how many times does the network need to run through the whole training set to train. Number of epochs also defines how long will the training run.

By normalizing the inputs learning process can be improved. The reason for it is because the gradients of the loss function with normalized values are symmetric, and greater learning rates can be used. Normalization can also apply to the outputs of the hidden layers, this is called Batch Normalization, as it applies to the batches, processed by training. Batch Normalization (BN) is used to speed up the training and regularize the model. Also, BN decreases the importance of initial randomly created weights.

It is also important to put a small data set aside, to test the model after each epoch, without training the parameters, but optimizing the hyperparameters as learning rate. This data set is called a validation set. In this work validation set is mostly taken as an independent data set, which is 5-20% of the size of the training set. The reason why the validation set is needed, is because with too many parameters, defined in the neural network, it can come to over-fitting. Over-fitting occurs, when after some epoch the loss of the trained data set is still decreasing, but a loss on the validation set is increasing. With too many parameters model fits too closely to a training data set, which can only be seen if results of the model are always applied to an external, not used for training, data set, validation set.

So far only the simplest (dense) neural network has been presented. The idea of it applies to almost all the networks, but the architecture and the specific functions differ. That is why it is always important to know exactly which data is processed by a network and what is the expected output. Dense neural network would work well with one-dimensional arrays but for two- or three-dimensional arrays (e.g. pictures) other tools, as Convolutions, are more convenient. For for more complex input data, like graphs, implementation of EdgeConv from [8] is more reasonable.

3.2 Convolutional Layers and Poolings

Convolution is a mathematical operation on two functions that produces a third function, that expresses how the shape of one is modified by the other. Although a fully connected layer can be used to learn the features of the input data and classify it, it would require an enormous amount of computation for inputs like pictures. With a picture size 40x40, as in this work, first hidden layer would get 1600 weights for each neuron (node of the hidden layer). In contrast to it, Convolutional Layers reduce the amount of parameters and allow the network to be deeper, as some of the parameters are shared along multiple nodes. This means that less computation is necessary and we get higher efficiency out of the model. Furthermore, convolutions are more more efficient in terms of memory usage, as they store less parameters.

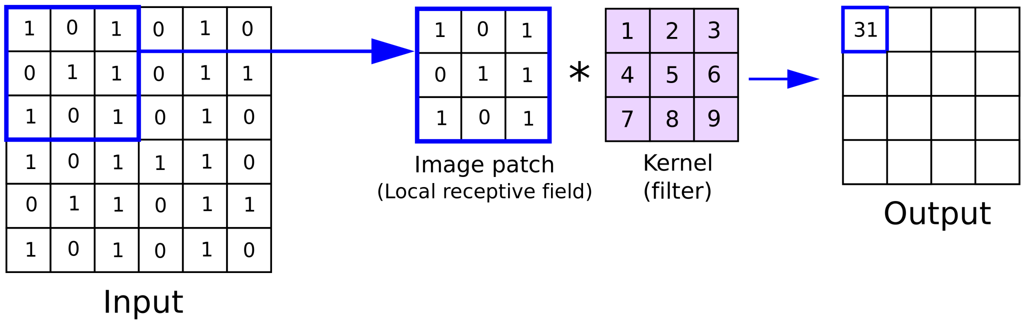


Figure 3.2: Example of a convolutional layer from reference [19]

As mentioned before, convolutional layers show high performances for image recognition tasks. That is why they will be essential for chapter 4. The main operation is a matrix multiplication between an image patch and a kernel. A learnable kernel, also called a filter, defined with some specific kernel size, e.g. 2x2 or 3x3, includes trainable weights that are multiplied with an image patch, as shown in the Fig. 3.2. Each point of the output is a multiplication between a filter and corresponding image patch. The filter slides from one image patch to the next. Therefore, the shape will contract if no padding is applied. If padding is activated zeros are added around image borders to conserve the spatial dimensions of the input. A Convolutional Layer is defined by the size and the number of filters, applied to an input.

Another layer, that is used in neural networks based on convolution, is a pooling layer. Pooling layers task is to reduce the dimension of the input by combining the outputs of neuron clusters at one layer into a single neuron in the next layer. There are different types of pooling, most commonly used are $Maximum-$ and $AveragePooling$, returning the maximum and average value of an image patch, respectively.

3.3 Edge Convolutoinal Layer

Edge Convolution idea was first presented in Dynamic Graph CNN [8]. It found huge success in a lot of different scientific branches, as well as physics [6], [10]. It operates on point clouds, where each point is described by some specific features.

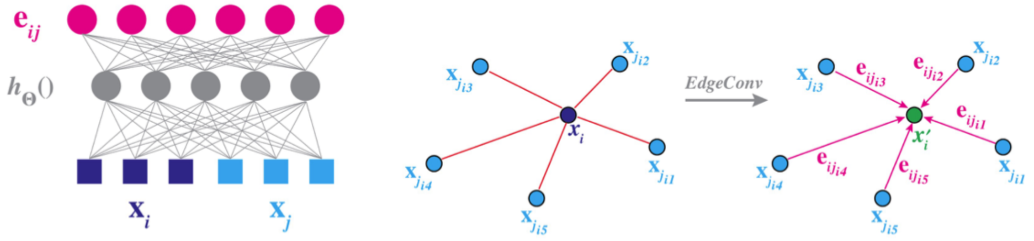


Figure 3.3: Example of an EdgeConv applying to the local patch in a point cloud. Picture from reference [8]

In the same way as Convolution operates on pictures applying kernels to local patches of those pictures, EdgeConv applies "edge kernel" to the "points patches". Consider input of one event as $\{x_1, x_2, ..., x_n\}$ where $x_i \subseteq \mathbb{R}^n$ is a point and n stands for the number of features of the point, in three-dimensional space n is three with features being coordinates (x, y, z). Point cloud is represented through a graph, in which vertices are points and edges correspond to the k nearest neighbouring connections between those points. EdgeConv operation to each point x_i is

$$x_i' = \square_{j=1}^{k} h_\Theta(x_i, x_j), \tag{3.8}$$

where x_i' corresponds to an output of an edge convolution in the point x_i. The edge function

$$h_\Theta(x_i, x_j) = \bar{h}_\Theta(x_i, x_j - x_i) \tag{3.9}$$

combines the particles features (x_i) with their local neighbourhood information (x_j) and applies a dense network and calculates edge features for each of its k neighbors. Instead of (x_i, x_j) we use a difference between the features of neighbours $(x_i, x_i - x_j)$, as this directly combines a global shape structre of the "point patch". $\bar{h}_\Theta$ can be implemented as a multilayer perceptron (MLP), whose learnable parameters are shared among all points, i.e. convolution. $\square$ is an aggregation operation, i.e. mean, max or sum. In this work we use mean as in ParticleNet [10].

Picture 3.3 shows, how EdgeConv applies to a local patch of particles with 5 neighbours, with x_i - particle, $x_{j1,2,3,4,5}$ - neighbours and x_i' - output of edge convolution for a particle.

The output of the edge convolution is as well a point cloud with the same number of points, having a changed dimension of features for each point. That is why edge convolutions can be stacked on each other the same as convolutions.

In our work we apply edge convolution to a graph of particles, that are decaying into new particles. This kind of graph is called a tree (construction of those tree-graphs is described further in the section 5.1). The intention to use the edge convolution on those tree-graphs is to bring more physics into the training process. While the convolutional layer applies to the final state of a measurement, hence particles on the picture, edge convolution aims to extract more information from the states of each emission.

3.4 Performance Measurement

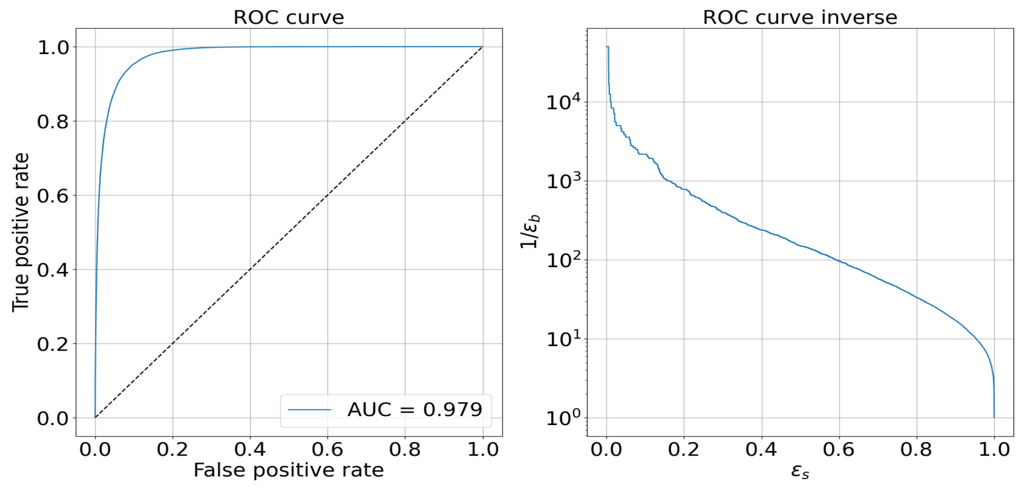


Figure 3.4: ROC curve example. On the left true positive rate plotted against false positive rate, with area under curve (AUC) in label section. On the right $1/\varepsilon_b$ is plotted against ε_s to better discrimination between different ROC-curves.

Performance of a model in classification task can be measured by the accuracy of tagging the inputs. Accuracy is a ratio between correctly classified data to all data, taken for a classification task. Along with accuracy comes loss, which is calcultaed by the loss function, defined for the task. Observing accuracy and loss of training and validation set can prevent the model from overfitting, as already discussed in section 3.1. The best score for accuracy is 1 corresponding to 100%, meaning, that all inputs, given to a model, were classified correctly.

While accuracy is a valid parameter to measure a performance of the model it doesn't represent all the relative qualities of the trained network. Softmax function (equation 3.6) at the end of the classification model gives the probability of an input belonging to a class. A class with the highest probability is assigned to the input. This implies, that in the model, that classifies data to 2 classes, inputs with output probabilities [0.4, 0.6] and [0, 1] would be assigned to the same class.

That is where receiver operating characteristic (ROC) curve becomes irreplaceable. ROC curve plots signal efficiency (ε_s), or true positive rate, and background efficiency (ε_b), or false positive rate, for all thresholds between 0 and 1, while accuracy is always calculated with a threshold 0.5 for 2-class-model. The picture on the left in the figure 3.4 is a common way of representing ROC-curves. Area under curve (AUC) is equal to the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one. The higher the AUC the better the classificatoin model is. A near-perfect diagnostic test would have the ROC curve that is almost vertical from (0,0) to (0,1) and then horizontal to (1,1), with AUC equal to 1. The black line in the middle is an example of a classification which randomly guesses classes of the model.

Right picture in the figure 3.4 represents the background rejection at a given signal efficiency.

This way of representing ROC curves is more appropriate for comparing models in this work, because a logarithmic scaling gives bigger gaps between models, than plotting false and true positive rate as on the left picture. This evokes higher visibility of the differences between models and therefore is more suitable to the comparison between them.

Another parameters used as comparison between two models in this work are background rejections at given signal efficiencies. In this way a concrete value from the right picture of the Fig. 3.4 is shown for different models. In this work we use background rejection at signal efficiencies of 30% and 50%, or $\varepsilon_s = 0.3$ and $\varepsilon_s = 0.5$.

4 CNN classifier

4.1 Jets as images

The Convolutional neural network in this work operates on calorimeter images, which were generated from 4-vectors. A jet, stored as list of 4-momenta, is transformed to the image by ploting the rapidity η against azimutal angle ϕ of each constituent of this jet. Limits, chosen for the pictures are $\eta, \phi \subset [-0.8, 0.8]$, as jets are produced with radius $R = 0.8$ in jet clustering algorithm. Some examples of images were already shown in section 2.

The backend, used in this work, is TENSORFLOW and KERAS 2.6.0. Networks are trained with the Adam optimizer [18] in it's default configuration if not declared otherwise.

In this work we present a Convolutional Neural Network architecture and the corresponding results in comparison to results in references [9] and [1].

4.2 Architecture

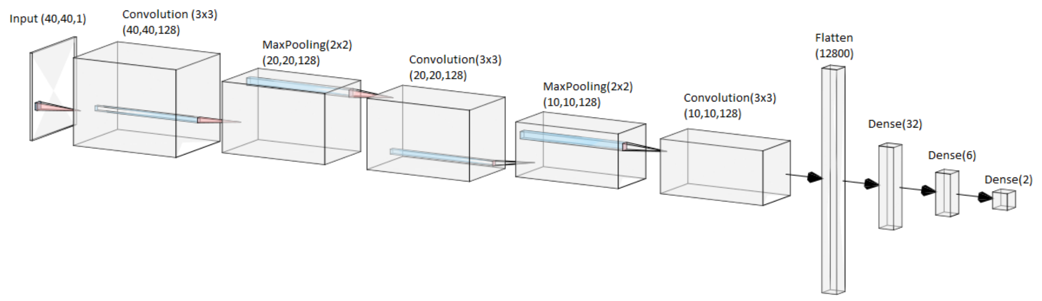


Figure 4.1: CNN architecture used in this reference. Architecture adapted from Ref. [9]

The convolutional neural network uses several convolutions for feature extraction and pooling operations for dimensionality reduction, followed by fully connected layers to a classification. This work uses a similar architecture as in Ref. [9], which is shown in Fig. 4.1.

The input is a picture with 40x40 pixels with depth 1. The convolutional layers use 128 filters with a kernel size 3x3 and ReLU (equation 3.5) as activation function. Two poolings with a 2x2 kernel, applied in between of the three convolutional layers, serve to reduce the dimensionality to 10x10 pixels, which are flattened as input to 3 fully connected dense layers. In the final dense layer there are two nodes that that correspond to 2 classes, to which the pictures are classified. Before the last step of classification a softmax (equation 3.6) activation function is applied, which gives a probability of the input belonging to a class.

The network is trained on 200k background and 200k signal images. Validation data consists of extra 20k background and 20k signal images. We train for 50 epochs. Loss is calculated with the categorical cross entropy function as discussed in the section 3.1. Learning rate is set to 0.001 and is decreased each 5 epochs by the factor 0.1 until the minimum value of 0.000001 is reached, if validation loss doesn't fall. If after 10 epochs validation loss does not improve the training process is stopped and the model is saved. This is called early stopping. In this way overfitting is prevented.

4.3 Results

After training, the model is tested on 100k background and 100k signal images, that are not used in training non validation. Model was trained 5 times to evaluate the training stability. In the figure 4.2 an example of loss and accuracy development throuhout one of those trainings is shown.

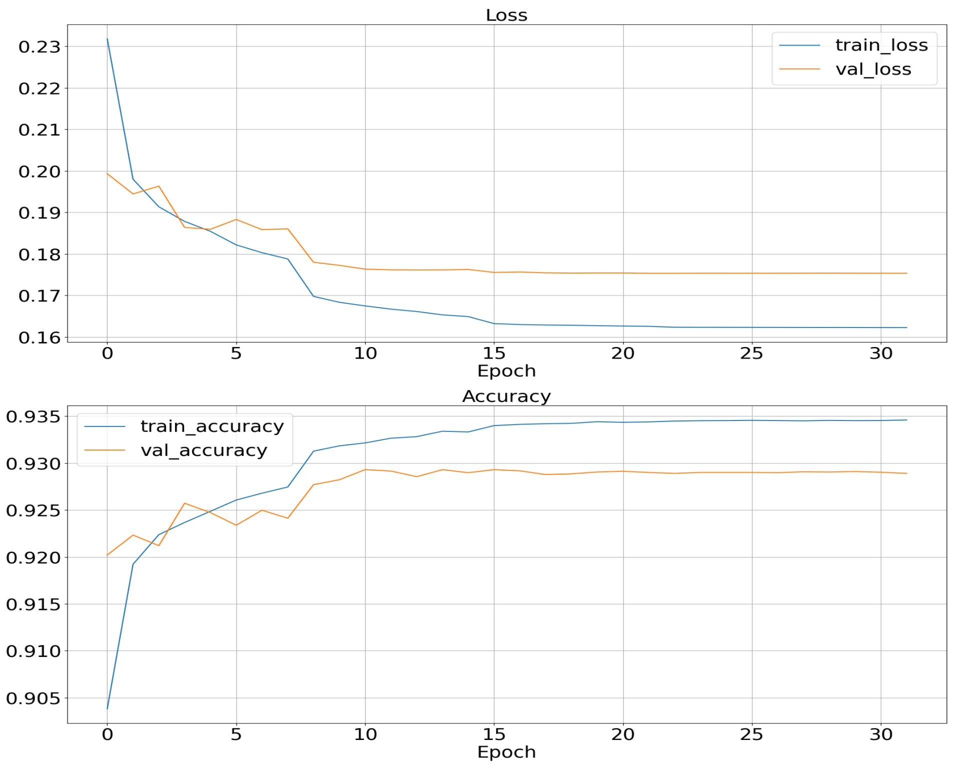


Figure 4.2: CNN loss and accuracy

Fig. 4.2 shows, how validation loss and accuracy behave in comparison to training loss and accuracy. Looking at the upper panel, the training loss decreases more steadily than validation loss, because after first epochs weights do not have enough updates to perform well on yet unseen data. Nevertheless the validation loss fluctuation drops with more epochs.

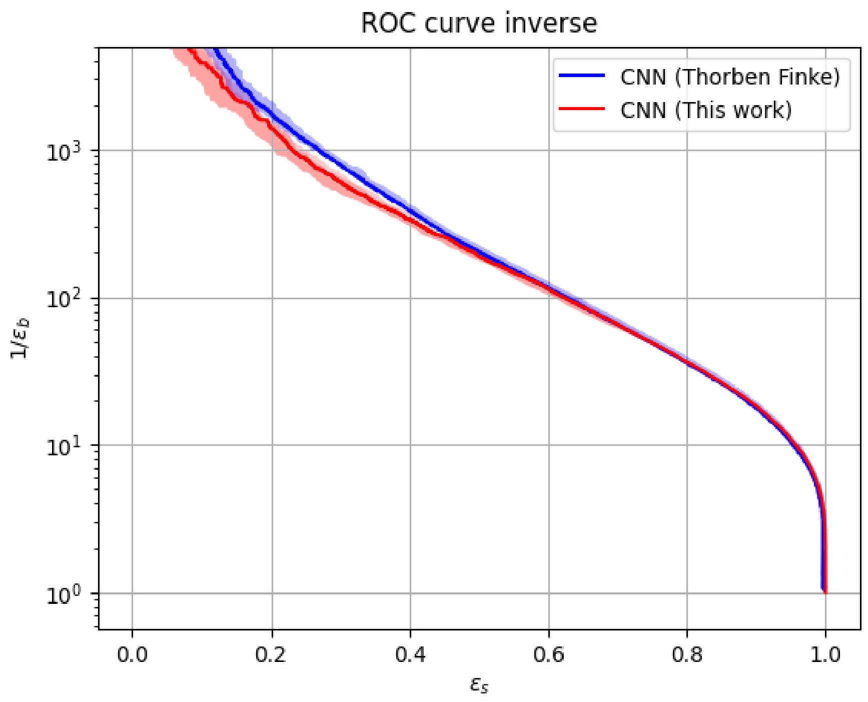


Figure 4.3: CNN ROC curve in comperison to the ROC curve from reference [9]

Figure 4.3 represents ROC curves of CNN modele from this work (red line) and CNN model from reference [9] (red line). Bands with slightly less intensity correspond to the spread of all five trainings. A comparison of loss, accuracies, AUC and background rejection at the signal efficiencies 30% and 50% between those two models are presented in the table 4.1.

It can be seen that results of Thorben Finke and this work lie very close to each other, with model from reference [9] giving a slightly better performance. Both ROC curves behave similar until the signal efficiency of 50%, afterward they split and their values at 30% differ a bit, as shown in Tab. 4.1. Nevertheless both curves lie very close to each other and a gap between them can be overcome by training the model on more data, which is shown in appendix A.2.

Table 4.1: CNN model results comparison

	This work	Thorben Finke [9]
Mean Loss	0.1780 ± 0.0033	–
Mean Accuracy (%)	92.91 ± 0.09	92.92 ± 0.05
AUC	0.9799 ± 0.0005	0.9802 ± 0.0002
$1/\varepsilon_b$ ($\varepsilon_s = 0.3$)	602.0 ± 66.1	785.7 ± 24.6
$1/\varepsilon_b$ ($\varepsilon_s = 0.5$)	189.4 ± 9.0	–

5 LundNet classifier

The idea of using Lund Plane graph structures for training the models brought a lot of exciting results to jet tagging problem [6]. It was first presented in reference [5]. In this chapter we introduce the Lund Plane and how the Lund Plane is constructed. Afterwards, we demonstrate neural networks that perform on Lund planes.

5.1 Data-preparation

Simulated jets, or sometimes called events, consist of particles 4-vectors. Each event, simulated as described in chapter 2, contains up to 200 particles. Those particles are firstly reclustered with Cambridge/Aachen algorithm into a tree-graph (section 5.1.1) and then the Lund coordinates of the reclustered particles are calculated (section 5.1.2). In this way the Lund plane is constructed for each event, which afterwards is used as an input into a neural network.

5.1.1 Declustering

Jet reconstruction is an approach to find unobserved partons, that after hadronization and fragmentation created new colourless particles, that are obtained at the detector. By reversing quantum mechanical process of fragmentation and hadronization a particle shower after the collision can be recreated. Jet finding is not a unique procedure, there are different jet algorithms, e.g. the k_t, the anti-k_t or the Cambridge/Aachen algorithm, all of them are presented in Ref. [15].

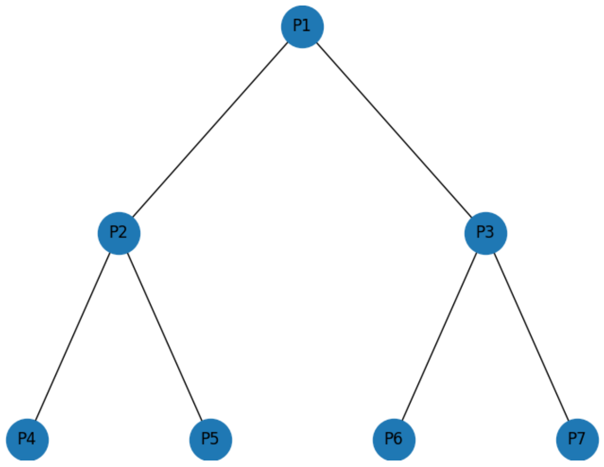


Figure 5.1: Declustered tree of particles

The Cambridge/Aachen algorithm is based on the k_t-algorithm with the difference, that Cambridge/Aachen recombines particles in order of their smallest angular difference R_{ij} (equation 5.1) and k_t in order of their smallest transverse momenta $p_t = \sqrt{p_x^2 + p_y^2}$. To recombine all the particles of a jet, Cambridge/Aachen recombination algorithm is used in this work, because of physical properties of the C/A algorithm and associated higher-order perturbative structures, described in more details in Ref. [5].

The definition of Cambridge/Aachen algorithm is as follows:

1. For each particle, the rapidity $\eta_i = \frac{E_i + p_{z,i}}{E_i - p_{z,i}}$ and the azimuthal angle ϕ_i are calculated from their 4-momenta. The distances $d_{ij} = R_{ij}^2/R^2$, where

$$R_{ij}^2 = \Delta\eta_{ij}^2 + \Delta\phi_{ij}^2 \tag{5.1}$$

for each pair of particles, are calculated. R is a jet-radius parameter, which in our work is set to 1000.

2. Find the minimum d_{min} of all the d_{ij}. Particles i and j, which d_{min} corresponds to, are merged, i.e. their four-momenta are added, if $d_{min} < d_{cut}$. If $d_{min} > d_{cut}$ then particles have to be stored as final jets.

3. Repeat steps 1 and 2 until no particles are left.

A visualization of an output from jet-algorithm is presented in the Fig.5.1. The C/A takes particles P4, P5, P6 and P7 as an input and recombines them to P2 and P3, which are afterwards recombined in P1. Particles with smallest R_{ij}^2 are clustered into jets, until all distances $R_{i,j}^2/R^2$ between jets are above some pre-defined value d_{cut}, which is only used if particles are not reclustered to the first particle, node P1 in the Fig. 5.1. In our work we need the reclustering to the node P1 for each jet, that is why d_{cut} is not used.

For better undersatanding of reclustering algorithms we construct the k_t algorithm, as it has the same idea as C/A with a slightly simpler implementation, with Python 3.6 and compare the results of the reclustering by this algorithm in comparison with the k_t algorithm from `FastJet` 3.4.0 [15], tested on the same events. We get same results for both of the implementations, the results are shown in the appendix A.3. Further only `FastJet` 3.4.0 is used in this work, as it is more user-convenient for extracting every point of the reclustered particle shower.

5.1.2 Lund Plane

After the C/A algorithm we get an output in a shape of the tree as in Fig. 5.1, where each node contains the 4-momenta of the particles. The inspiration behind using Lund Planes as an input to jet tagging task is because their features give broader physical information about the emissions, than the 4-momenta of the emitted particles.

Figure 5.2 shows three different representations of two jets. Pictures at the top represents a jet (a), decaying in jets (b) and (c), with either jet (c) being an emission of jet (a) (left part of the picture) or jet (b) (right part of the picture). Picture in the middle represents a Lund diagram in a phase-space triangle with coordinates $ln(k_t)$ and $ln(\Delta)$, where k_t is a transverse momentum

$$k_t = p_{t,b}\Delta_{ab} \tag{5.2}$$

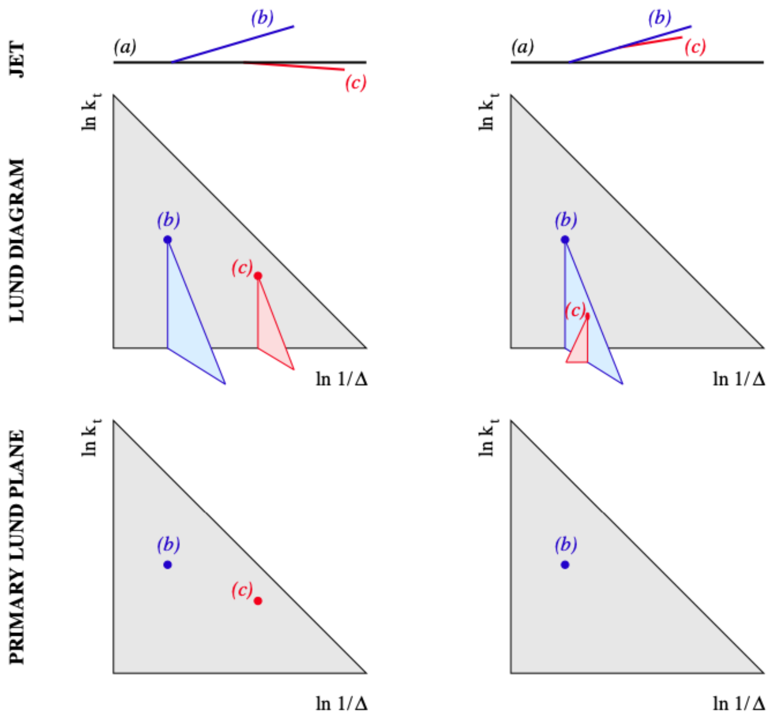


Figure 5.2: Lund plane. Picture from reference [6]

and Δ_{ab} an angle of an emitted particle with respect to its emitter

$$\Delta_{ab}^2 = (\eta_a - \eta_b)^2 + (\phi_a - \phi_b)^2 \tag{5.3}$$

with $p_{t,b} < p_{t,a}$, with a then called a softer branch and b a herder branch. Lund diagrams are triangle-shaped, because this representation allows to easily read features, such as mass, angle and momentum of the emission. The bottom picture shows the primary Lund plane, which only contains the emissions of the primary (hardest) jet.

Lund plane contains a wide information about decays of the jet, that is why Lund plane tuples are taken as input features in a neural network architecture in this work. The C/A algorithm is the main tool of constructing the Lund plane out of jet constituents. With C/A algorithm we recombine all particles in the event, saving reclustered pseudojets as tuples $T^{(i)}$ with

$$T^{(i)} = \{ln(k_t), ln(\Delta), ln(z), ln(m), ln(\psi)\}. \tag{5.4}$$

Variables k_t and Δ are from the equations 5.2 and 5.3, $z = p_{t,b}/(p_{t,a} + p_{t,b})$ is the momentum fraction of the softer subjet b, m is the mass of the emitter (a and b together) and ψ =arctan$(\frac{\eta_b - \eta_a}{\phi_b - \phi_a})$.

After Lund coordinates are calculated we construct a graph with features corresponding to the Lund coordinates. We represent graphs as a combination of three matrices: *features*, *neighbours* and *mask*. *Neighbours* include all the neighbourhood connections of each node, while *features* are

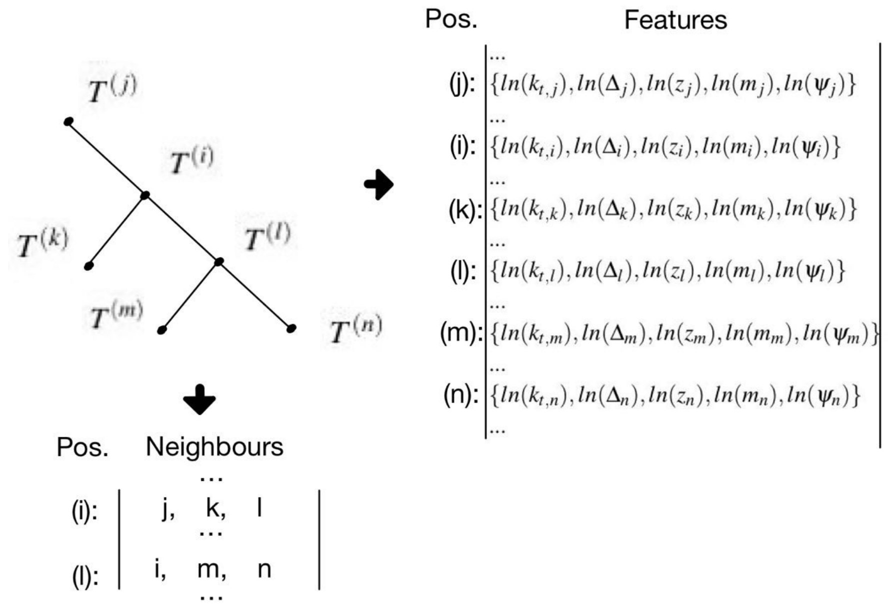

Pos.
Features
(j): $\{ln(k_{t,j}), ln(\Delta_j), ln(z_j), ln(m_j), ln(\psi_j)\}$
(i): $\{ln(k_{t,i}), ln(\Delta_i), ln(z_i), ln(m_i), ln(\psi_i)\}$
(k): $\{ln(k_{t,k}), ln(\Delta_k), ln(z_k), ln(m_k), ln(\psi_k)\}$
(l): $\{ln(k_{t,l}), ln(\Delta_l), ln(z_l), ln(m_l), ln(\psi_l)\}$
(m): $\{ln(k_{t,m}), ln(\Delta_m), ln(z_m), ln(m_m), ln(\psi_m)\}$
(n): $\{ln(k_{t,n}), ln(\Delta_n), ln(z_n), ln(m_n), ln(\psi_n)\}$
$T^{(j)}$ $T^{(i)}$ $T^{(k)}$ $T^{(l)}$ $T^{(m)}$ $T^{(n)}$
Pos. Neighbours
(i): j, k, l
(l): i, m, n

Figure 5.3: Visualization of how graphs of tuples $T^{(i)}$ are stored in *neighbours* and *features*

tuples $T^{(i)}$ containing Lund coordinates of each node. A visualiuation of how the graph patch is represented in *neighbours* and *features* is shown in the Fig. 5.3.

Each node in a tree-graph (Fig. 5.3 top left picture) can have maximum three connections, like tuples $T^{(i)}$ and $T^{(l)}$. The number of each neighbour is stored in the *neighbours* in the respective position, e.g. for the tuple $T^{(i)}$ at the position (i) numbers j, k, and l are stored. As the maximum number of particles in each event is 200, maximum number of nodes, that a reclustered tree-graph can have, is 199. For the case if some nodes have less than three connections an additional node with zero values in *features* is created at the end of each graph (200th node). This node serves as a reference for every non existing connection, e.g. node with two connections has in the *neighbours* those two connections and 200th node as the third connection.

Number of particles in each event is not fixed, which implies that number of nodes in a graph differs as well. That is why *mask* is used as a tag for existing and non existing nodes.

`FastJet 3.4.0` is used in this work for C/A jet reclustering. Constructed Lund Planes are firstly represented through graphs from `NetworkX 2.5.1` and then transformed to *neighbours*, *features* and *mask* as described in section 5.1.2. This three matrices represent the Lund planes that are constructed from 4-vectors of particles of each jet. They are used as input into neural networks, described in sections 5.2 and 5.3.

5.2 LundNet architecture

In contrast to the network in the reference [6], which used PyTorch 1.7 and Deep Graph Library 0.4.3 as backend, the LundNet in this work uses TensorFlow 2.6.0 and Keras 2.6.0 and is adjusted from ParticleNet implementation [10]. In the following, the TensorFlow implementation is refered

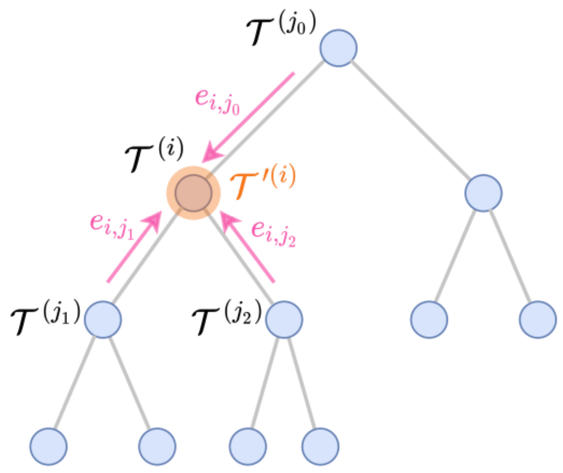


Figure 5.4: Visualization of Edge Conv operation on a node. Picture from reference [6]

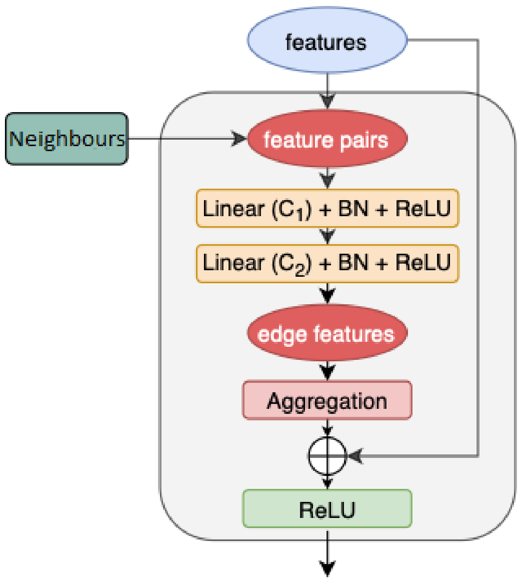


Figure 5.5: EdgeConv layer used in a LundNetTf model, modified from reference [6]

to as LundNetTF.

The idea of graph network is adapted from LundNet [6], with edge convolution from Dynamic Graph CNN [8] being a key step. LundNetTF is build on edge convolutional layers, described in chapter 3. Visualization of how edge features apply to nodes of a reclustered lund tree is shown in figure 5.4.

Figure 5.5 shows the architecture of the EdgeConv block, used in this work. As shown in the Fig. 5.5, first a multi-layer perceptron (MLP) (equation 3.9) is applied to feature pairs of all incomming nodes, using *neighbours* to find all the pairs. As shown in the figure 5.5, MLP consists of two layers, each of them consists of a linear layer, with C1 and C2 number of filters and linear layer implemented as a convolution with kernel size 1x1, followed by a batch normalization, and a ReLU activation function. Then, the aggregation step takes an element-wise average of the learned edge features of all the incoming edges. Afterwards, a global ReLU activation is applied to the upgraded features. This operation performs on all nodes using the same MLP without changing the graph structure, which is described by the *neighbours*.

The LundNetTF architecture is shown in figure 5.6. We build neural network on six EdgeConv blocks as in reference [6]. Filters, given to the edge convolution, are (32, 32), (32, 32), (64, 64), (64, 64), (128, 128) and (128, 128). After the edge convolution part a channel-wise global average pooling operation is applied to aggregate the learned features over all nodes. This is followed by a fully connected layer with 256 units and using a dropout rate 0.1 with ReLU activation function. In the end a fully connected layer with two units and softmax activation function serves as classification output.

The network is trained on 200k background and 200k signal events. Validation data consists of extra 10k background and 10k signal events. We train for 50 epochs. Loss is calculated via

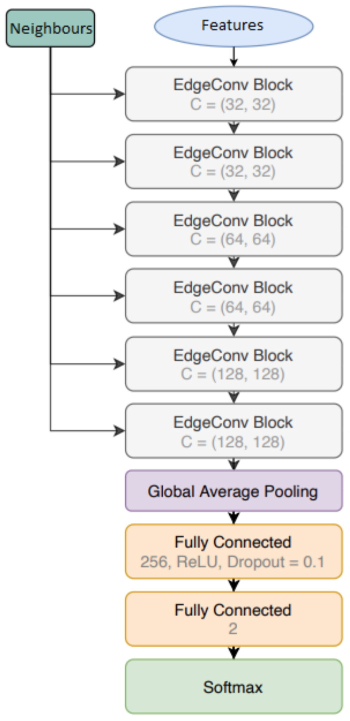


Figure 5.6: LundNetTF architecture. Picture modified from reference [6]

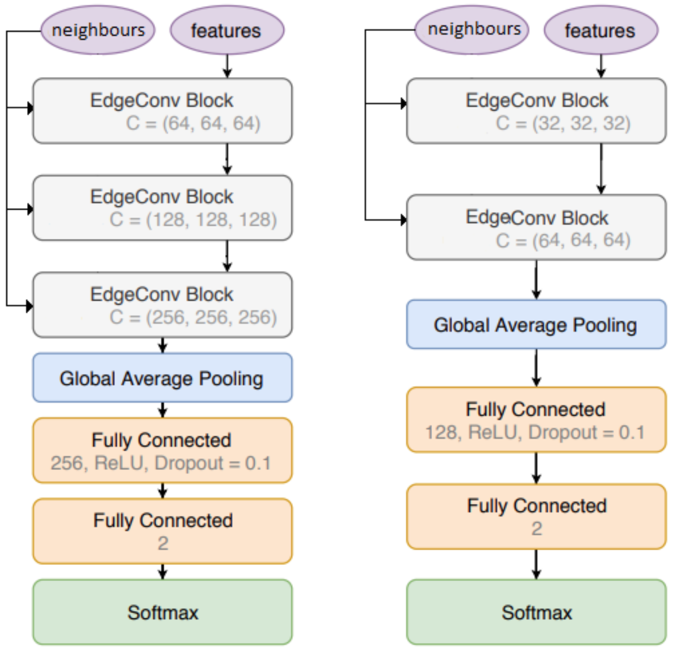


Figure 5.7: NeighboursNet architectures. Picture modified from reference [10]

categorical crossentropy. Learning rate is optimized via Adam [18] optimizer. If after 10 epochs validation loss does not improve training will be stopped.

5.3 NeighboursNet architecture

In this work, we also test the ParticleNet[10] architectures shown in figure 5.7, with a difference that knn-search can be skipped, same as in the LundNetTF idea, because we already constructed a graph that contains all the connections. In this work we refer to this networks as NeighboursNet and NeighboursNetLite, as they are adopted from ParticleNet and ParticleNetLite.

NeighboursNet is shown in the figure 5.7 on the left. It consists of three EdgeConv blocks. EdgeConv block in this architecture is similar to the edge convolution in section 5.2 with a difference, that it has one additional layer consisting of linear layer, batch normalization and ReLU activation function. The number of filters for each EdgeConv block is (64, 64, 64), (128, 128, 128), and (256, 256, 256), respectively. This construction gives a bigger amount of trainable parameters than in LundNetTF. After the EdgeConv block a global average-pooling operation is applied to combine the learned features. After this a fully connected layer with 256 units, ReLU activation function and dropout rate 0.1 is applied, before a final classification.

NeighboursNetLite architecture is shown in the figure 5.7 on the right. This simplified architecture uses the same edge convolution as the first, but is less complex than the first one and the

LundNetTF. Compared to the first NeighboursNet architecture it uses only two EdgeConv blocks and number of channels is reduced to (32, 32, 32) and (64, 64, 64) for the two blocks, respectively. Amount of units in the dense layer is as well shortened to 128. The number of calculations is lowered, because the number of trainable parameters is vastly lower than in NeighboursNet and LundNetTF.

All networks are trained on the same 200k signal and 200k background events, as in the section 5.2. Validation data is as well 10k background and signal events each. Same as in section 5.2, models are trained for 50 epochs with a callback to stop training after 10 epochs without validation loss improvement.

5.4 Results

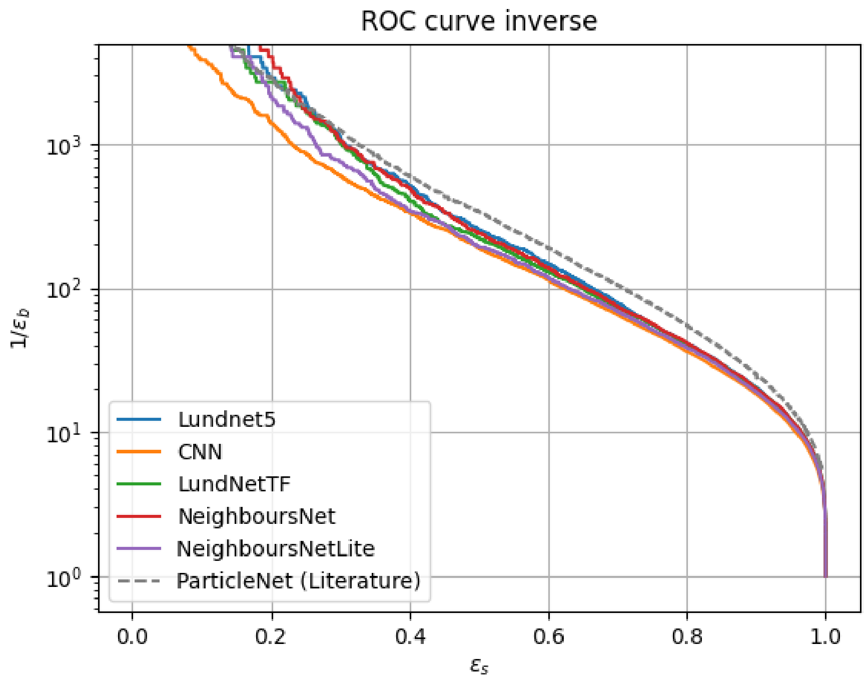


Figure 5.8: Results of background rejection against top tagging in this work

The LundNet5 neural network from reference [6], using the PyTorch backend, is available on GitHub [20]. For this work it is trained on the same data as LundNetTF. The number 5 in the name LundNet5 stands for the number of Lund coordinates used as features. While in this work we only concentrate on tuples $T^{(i)}$ with 5 coordinates, reference [6] provides as well LundNet trained on reclustered tuples containing only the first three coordinates, i.e. $(ln(k_t), ln(\Delta), ln(z))$.

LundNet5 is trained on 200k background events and 200k signal events with 10k signal and background events each for validation. All models from chapter 5 are tested on 40k signal and background events each. In figure 5.8, results of LundNet5 is referred as "Lundnet5" and corresponds to the blue line. The green line corresponds to LundNetTF, developed in this work and described in section 5.2. Red and purple lines correspond to NeighboursNet and NeighboursNetLite described in section 5.3. All models are as well compared to CNN, trained in this work. Yellow line corresponds to the ROC curve of the CNN model from chapter 4, that results lie closest to the average from Table 4.1. Grey dashed line corresponds to the results of ParticleNet from reference [10].

Table 5.1 shows results of all models, trained in this work.

Achieved accuracy and AUC values of every model in this work are very high. Furthermore, it can be seen in the table 5.1, that NeighboursNet model delivers similar, probably slightly lower, results, as the implementation of LundNet5 from GitHub ref. [20] tested on the same data as NeighboursNet. All models in this work get lower background rejection than ParticleNet [10] at signal efficiencies over 30-40%, but come close to the ParticleNet ROC curve at lower signal efficiencies, especially NeighboursNet, LundNetTF and LundNet5.

Table 5.1: Comparison of all architectures

	NeighboursNet	NeighboursNetLite	LundNetTF	LundNet5	CNN
Loss	0.1665	0.1716	0.1676	-	0.1780
Accuracy (%)	93.36	93.11	93.34	93.39	93.02
AUC	0.9825	0.9811	0.9822	0.9825	0.9801
$1/\varepsilon_b$ ($\varepsilon_s = 0.3$)	1065.0	749.44	1037.69	1093.78	602.41
$1/\varepsilon_b$ ($\varepsilon_s = 0.5$)	240.18	194.10	221.15	251.37	187.97

6 Discussion and Conclusion

In this work we use deep learning for jet classification at the LHC. To find new physics Beyond Standard Model a search for anomalies, signals, became a very important task. Top quark plays a significant role in this search, as many extensions to Standard Model predict top partners which eventually decay into top quarks. That is why as a benchmark signal we use top jets. As background we use light QCD jets. We test two different approaches to classify those jets. First, a CNN is applied to the calorimeter images, generated from 4-vectors of jet constituents (chapter 4). Reconstructed particle showers includes more information about a jet than 4-momenta of its constituents. To apply more physics to the training process, Lund planes are constructed out of jet constituents, which are used as input to the LundNet (chapter 5).

In the section 4.3, we see, that CNN shows very high performance in classifying the images, that are constructed from 4-momenta of jet constituents. Accuracy is higher than 92% and AUC is approximately 0.98, which is very close to perfect. The stability of the training can be seen from the Fig. 4.3, which shows that ROC curves from this work and Ref. [9] lie very close to each other and both have only little fluctuations of all 5 trainings each. The numbers from table 4.1 show that the CNN in this work performed similar to the CNN in the reference [9] and the difference can be overcome by training the model on more data, which is shown in appendix A.2.

Section 5.4 shows results of the LundNetTF classifier described in sections 5.2 and two NeighboursNet classifiers from section 5.3. Achieved performances are very high, as the accuracies of the models are over 93% and AUC of each model is higher than 0.98. ROC curves of all models are presented in the figure 5.8. It can be seen that every model, that uses Lund planes as input, performs better than the CNN, delivering higher background rejection at almost every signal efficiency. For comparison we use the background rejection at signal efficieny $\varepsilon_s = 30\%$, which is lower than for any other model, trained on the Lund planes, as presented in the table 5.1.

Achieved results of the models from chapter 5 are higher than from most of the algorithms presented in the reference [1]. The only algorithms, that are surpassing the values of the LundNetTF, LundNet5 and NeighboursNet are ParticleNet [10] with the accuracy 93.8% and AUC 0.985 and ResNeXt [21] with the accuracy 93.6% and AUC 0.984. Nevertheless, the advantage of the algorithms, presented in this work, in comparison to ParticleNet and ResNeXt is a smaller number of parameters, which implies a smaller amount of necessary computations. While ParticleNet has approximately 500k and ResNeXt over 1.4M parameters, NeighboursNet has 370k and LundNetTF aproximately 180k.

Although LundNet-based networks deliver better results than the CNN, tagging can still be improved. It can be seen that NeighboursNet gets higher background rejection than LundNetTF, which means, that hyperparameters, like different combinations of filters in the EdgeConv and different stacking of EdgeConvs, influence the training process. Thus, by testing another combinations of those hyperparameters better results can be obtained.

Furthermore, another aspect that can be researched and upgraded is data preparation. Every event in this work constructs a Lund plane with a fixed maximum number of nodes (200), whereas only a small amount of jets reconstructs into a graph with this number of nodes, most of the times this number even lies under 100. Mask takes care of the rest of the graph and sets all non-existent features to zero. That implies that a big part of inputs into the neural network consists of empty data. One possible solution is to search for a perfect cutoff maximum numbers of nodes. However this can lead to the loss of some important data.

To summarize, there is a lot of data from LHC that can be analyzed to discover new beyond the standard model physics. By using deep learning it is possible to classify different jets, which can result into finding BSM. By using a reconstructed particle shower as an input into a neural network better performance as in convolutional neural networks can be achieved. This means that data preparation is as crucial for deep learning tasks as it's algorithms. Developing both of those aspects is a prominent way to improve jet tagging, which results into finding new, yet undiscovered landscape of physics.

A Appendix

A.1 CNN architecture from chapter 4

The implementation of all models in this work uses TENSORFLOW and KERAS standard layers. The following code corresponds to the convolutional neural network used in this work.

```
model = models.Sequential(
    [
        layers.Conv2D(128,
                      input_shape = (40,40,1),
                      kernel_size = (3,3),
                      strides = (1,1),
                      padding = "same",
                      dilation_rate = 1,
                      activation = "relu"),
        layers.MaxPooling2D((2,2),
                            strides = None,
                            padding = "valid"),
        layers.Conv2D(128,
                      input_shape = (20,20,128),
                      kernel_size = (3,3),
                      strides = (1,1),
                      padding = "same",
                      dilation_rate = 1,
                      activation = "relu"),
        layers.MaxPooling2D((2,2),
                            strides = None,
                            padding = "valid"),
        layers.Conv2D(128,
                      input_shape = (10,10,128),
                      kernel_size = (3,3),
                      strides = (1,1),
                      padding = "same",
                      dilation_rate = 1,
                      activation = "relu"),
        layers.Flatten(),
        layers.Dense(36),
        layers.Activation("relu"),
        layers.Dense(6),
        layers.Activation("relu"),
        layers.Dense(2),
        layers.Activation("softmax")
    ]
)
```

A.2 CNN results for 300k signal and background images

In this work we train all model with 200k signal and background events each. Figure A.1 shows ROC curves of CNN models trained for with 200k (red line) and 300k (green line) in comparison

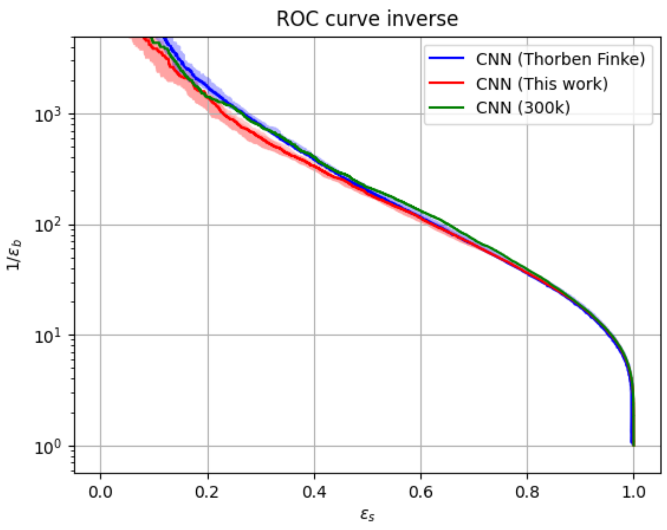


Figure A.1: CNN ROC curves in comperison to the ROC curve from CNN, trained with 300k background and signal images

to the ROC curve from reference [9].

It can be seen that CNN trained on 300k improves and even has slightly higher background rejection at signal efficiencies around 60%, but the improvement is not extreme in comparison to CNN trained on 200k.

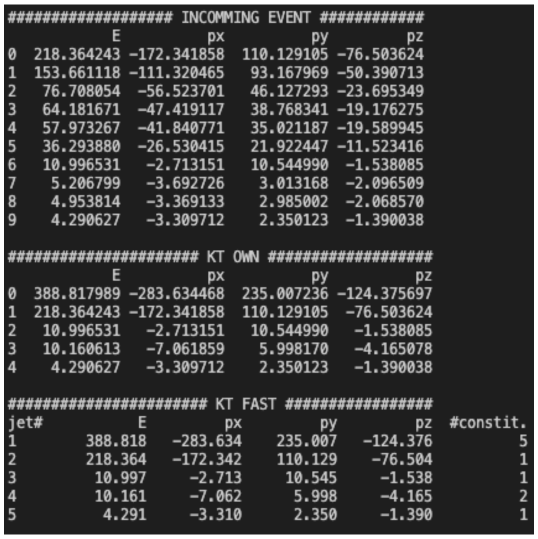
A.3 Jet declustering from section 5.1.1

In this work we test the k_t algorithm by constructing it with Python 3.6 (further referred as k_t-own) and comparing it to the k_t algorithms from `FastJet`-software [15]. Example of the results for the event with 10 particles is shown in the figure A.2.

A code for k_t-own can be found in our GitLab-repository [22].

The incomming event consists of 10 particles with 4-momentas, as shown in the figure A.2 under the line `INCOMMING EVENT`. The particles under the line `KT OWN` represent the particles, to which it was reclustred with the Radius $R = 0.05$ by the implementation of k_t-own. The particles under the line `KT FAST` represent the particles, to which it was reclustred with the Radius $R = 0.05$ by the `FastJet` implementation.

It can be seen the the reclustered particles of both algorithms contain identical 4-momenta, but the `FastJet` implementation allows as well to get the constituents of each of the reclustered particles, that is why this implementation is used in this work.



```
################### INCOMMING EVENT ############
            E          px          py          pz
0  218.364243 -172.341858  110.129105 -76.503624
1  153.661118 -111.320465   93.167969 -50.390713
2   76.708054  -56.523701   46.127293 -23.695349
3   64.181671  -47.419117   38.768341 -19.176275
4   57.973267  -41.840771   35.021187 -19.589945
5   36.293880  -26.530415   21.922447 -11.523416
6   10.996531   -2.713151   10.544990  -1.538085
7    5.206799   -3.692726    3.013168  -2.096509
8    4.953814   -3.369133    2.985002  -2.068570
9    4.290627   -3.309712    2.350123  -1.390038

###################### KT OWN ###################
            E          px          py           pz
0  388.817989 -283.634468  235.007236 -124.375697
1  218.364243 -172.341858  110.129105  -76.503624
2   10.996531   -2.713151   10.544990   -1.538085
3   10.160613   -7.061859    5.998170   -4.165078
4    4.290627   -3.309712    2.350123   -1.390038

####################### KT FAST #################
jet#            E          px          py          pz  #constit.
1         388.818    -283.634     235.007    -124.376          5
2         218.364    -172.342     110.129     -76.504          1
3          10.997      -2.713      10.545      -1.538          1
4          10.161      -7.062       5.998      -4.165          2
5           4.291      -3.310       2.350      -1.390          1
```

Figure A.2: Results of the reclustering of the incomming event via k_t implementation in FastJet [15] and this work

A.4 Lund plane construction from section 5.1.2

A complete jet declustering module can be found in the GitLab repository [22], but below is a small extract of the code, that shows an example of how FastJet-python implementation and NetworkX are used to construct a Lund plane in this work.

Firstly, we define a graph of a NetworkX-class and set Cambridge/Aachen-algorithm as jet definition. jets represent the reclustered pseudojets and with a function has_parents emissinos of the primary jet are stored in j1 and j2. Afterwards we calculate the Lund coordinates from j1 and j2 and store created tuple t1 as node. The same procedure applies to the first jet j1 and another node is created. Function add_edge creates a connection between two nodes in the graph. In this way a graph, including two first pseudojets of a reclustered event is constructed.

```python
g = nx.Graph()
jet_def = fj.JetDefinition(fj.cambridge_algorithm, 1000.0)
for p in event[1:]:
    constits.append(fj.PseudoJet(p['px'], p['py'], p['pz'], p['E']))
jets = jet_def(constits)
j1 = fj.PseudoJet()
j2 = fj.PseudoJet()
jets[0].has_parents(j1, j2)
if (j2.pt() > j1.pt()):
    j1, j2 = j2, j1
t1 = LundCoordinates(j1, j2)
g.add_node(0, features=t1) #node 0 is created
j1_1 = fj.PseudoJet()
j1_2 = fj.PseudoJet()
jets[0].has_parents(j1_1, j1_2)
if (j1_2.pt() > j1_1.pt()):
    j1_1, j1_2 = j1_2, j1_1
t2 = LundCoordinates(j1_1, j1_2)
g.add_node(1, features=t2) #node 1 is created
g.add_edge((0,1)) #connection between 0 an 1 is established
```

A.5 LundNetTF architecture from section 5.2

EdgeConv implementation in this work is modified from ParticleNet[10] and can be found in our GitLab repository [22]. Following code shows how LundNetTf architecture is build on EdgeConv and TENSORFLOW and KERAS standard functions.

```
num_points = 200
shapes = {"neighbours": (num_points, 3), "features": (num_points, 5), "mask": (num_points,
    1)}

neighbours = keras.Input(name='neighbours', shape=input_shapes['neighbours'])
features = keras.Input(name='features', shape=input_shapes['features'])
mask = keras.Input(name='mask', shape=input_shapes['mask'])

coord_shift = tf.multiply(199., tf.cast(tf.equal(mask, 0), dtype='float32'))  # make non-
    valid positions to 200th node

nghb = tf.add(coord_shift, neighbours)
fts = tf.squeeze(keras.layers.BatchNormalization()(tf.expand_dims(features, axis=2)), axis
    =2)
fts = EdgeConv(nghb, fts, num_points, (32, 32))
fts = EdgeConv(nghb, fts, num_points, (32, 32))
fts = EdgeConv(nghb, fts, num_points, (64, 64))
fts = EdgeConv(nghb, fts, num_points, (64, 64))
fts = EdgeConv(nghb, fts, num_points, (128, 128))
fts = EdgeConv(nghb, fts, num_points, (128, 128))
fts = tf.multiply(fts, mask)
x = tf.reduce_mean(fts, axis=1)
x = keras.layers.Dense(256, activation='relu')(x)
x = keras.layers.Dropout(0.1)(x)
outputs = keras.layers.Dense(2, activation='softmax')(x)

model = keras.Model(inputs =[neighbours , features , mask], outputs=outputs)
```

Listing A.1: LundNetTF

NeighboursNet and NeighboursNetLite are build same as LundNetTF, but with another parameters for filters in EdgeConv, as desctibed in section 5.3.

References

[1] G. Kasieczka, T. Plehn, A. Butter, K. Cranmer, D. Debnath, B. M. Dillon, M. Fairbairn, D. A. Faroughy, W. Fedorko, C. Gay, L. Gouskos, J. F. Kamenik, P. T. Komiske, S. Leiss, A. Lister, S. Macaluso, E. M. Metodiev, L. Moore, B. Nachman, K. Nordstrom, J. Pearkes, H. Qu, Y. Rath, M. Rieger, D. Shih, J. M. Thompson, and S. Varma, "The Machine Learning Landscape of Top Taggers", *SciPost Phys.* [online], vol. 7 1 2019, p. 14, 1 2019. DOI: `10.21468/SciPostPhys.7.1.014`. available from: `https://scipost.org/10.21468/SciPostPhys.7.1.014`.

[2] S. Höche, *Introduction to parton-shower event generators*, 2015. arXiv: `1411.4085 [hep-ph]`.

[3] Z. Nagy and D. E. Soper, What is a parton shower? *Physical Review D* [online], vol. 98, no. 1 Jul. 2018, Jul. 2018, ISSN: 2470-0029. DOI: `10.1103/physrevd.98.014034`. available from: `http://dx.doi.org/10.1103/PhysRevD.98.014034`.

[4] B. Graham, Spatially-sparse convolutional neural networks, *CoRR* [online], vol. abs/1409.6070 2014, 2014. arXiv: `1409.6070`. available from: `http://arxiv.org/abs/1409.6070`.

[5] Frederic A. Dreyer, Gavin P. Salam, Gregory Soyez. (Oct. 2018). “The Lund Jet Plane.” arXiv: `1807.04758v2`.

[6] Frédéric A. Dreyer, Huilin Qu. (Feb. 2021). “Jet tagging in the Lund plane with graph networks.” arXiv: `2012.08526v2`.

[7] Yu.L. Dokshitzer, G.D. Leder, S. Moretti, B.R. Webber. (Jul. 1997). “Better Jet Clustering Algorithms.” arXiv: `hep-ph/9707323v2`.

[8] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, Justin M. Solomon. (Jan. 2018). “Dynamic Graph CNN for Learning on Point Clouds.” arXiv: `1801.07829v2`.

[9] Thorben Finke, Deep Learning for New Physics Searches at the LHC 2020, 2020.

[10] Huilin Qu, Loukas Gouskos. (Mar. 2020). “ParticleNet: Jet Tagging via Particle Clouds.” arXiv: `1902.08570v3`.

[11] D. E. Morrissey, T. Plehn, and T. M. Tait, Physics searches at the lhc, *Physics Reports* [online], vol. 515, no. 1-2 May 2012, pp. 1–113, May 2012, ISSN: 0370-1573. DOI: `10.1016/j.physrep.2012.02.007`. available from: `http://dx.doi.org/10.1016/j.physrep.2012.02.007`.

[12] J. Alwall, R. Frederix, S. Frixione, V. Hirschi, F. Maltoni, O. Mattelaer, H.-S. Shao, T. Stelzer, P. Torrielli, and M. Zaro, The automated computation of treelevel and next-to-leading order differential cross sections, and their matching to parton shower simulations, *Journal of High Energy Physics* 2014, 2014.

[13] T. Sjöstrand, S. Ask, J. R. Christiansen, R. Corke, N. Desai, P. Ilten, S. Mrenna, S. Prestel, C. O. Rasmussen, and P. Z. Skands, An introduction to pythia 8.2, *Computer Physics Communications* [online], vol. 191 2015, pp. 159–177, 2015, ISSN: 0010-4655. DOI: `https://doi.org/10.1016/j.cpc.2015.01.024`. available from: `https://www.sciencedirect.com/science/article/pii/S0010465515000442`.

[14] J De Favereau, Christophe Delaere, Pavel Demin, Andrea Giammanco, Vincent Lemaitre, Alexandre Mertens, Michele Selvaggi, Delphes 3 Collaboration, et al., Delphes 3: A modular framework for fast simulation of a generic collider experiment, *Journal of High Energy Physics* 2014, 2014.

[15] Matteo Cacciari, Gavin P. Salam, Gregory Soyez. (Nov. 2011). "FastJet user manual." arXiv: `1111.6097`.

[16] Klemradt Uwe, Deep Learning in Physics Research 2020, 2020.

[17] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[18] Diederik P. Kingma, Jimmy Ba. (Jan. 2017). "Adam: A Method for Stochastic Optimization." arXiv: `1412.6980`.

[19] Anh H. Reynolds. (2019). "Convolutional Layer," available from: `https://anhreynolds.com/blogs/cnn.html`.

[20] (2021-01-15). "LundNet GitHub," available from: `https://github.com/fdreyer/LundNet`.

[21] S. Xie, R. B. Girshick, P. Dollár, Z. Tu, and K. He, Aggregated residual transformations for deep neural networks, *CoRR* [online], vol. abs/1611.05431 2016, 2016. arXiv: `1611.05431`. available from: `http://arxiv.org/abs/1611.05431`.

[22] (2021-09-15). "Kostiantyn Lavronenko GitLab," available from: `https://git.rwth-aachen.de/lavronenkostya/bt`.