

# Deep learning for new physics mining at the LHC

*Simon Klüttermann*

MASTER THESIS IN PHYSICS

submitted to the

FACULTY OF MATHEMATICS COMPUTER SCIENCE AND  
NATURAL SCIENCES

RWTH AACHEN UNIVERSITY

DEPARTMENT OF PHYSICS

INSTITUTE FOR THEORETICAL PARTICLE PHYSICS AND  
COSMOLOGY

First Referee: Prof. Dr. Michael Krämer  
Second Referee: Prof. Dr. Felix Kahlhöfer

November 2020



# Table of content

<b>Table of content</b>	<b>1</b>
<b>1 Motivation</b>	<b>3</b>
<b>2 Introduction and literature</b>	<b>5</b>
2.1 New physics . . . . .	5
2.2 Neuronal networks and autoencoder . . . . .	5
2.3 Graphs . . . . .	8
2.4 Graph autoencoder . . . . .	10
<b>3 Basic concepts</b>	<b>11</b>
3.1 Binary classification . . . . .	11
3.2 Datapreperation . . . . .	14
3.3 Explaining figures used in this thesis . . . . .	15
<b>4 A working graph autoencoder</b>	<b>18</b>
4.1 Graph neural networks . . . . .	18
4.2 The compression algorithm . . . . .	19
4.3 the decompression algorithm . . . . .	19
4.4 Our model setup . . . . .	20
4.5 Choosing the righth loss . . . . .	21
4.6 Difficulties when evaluating a model . . . . .	24
4.7 Evaluating the autoencoder . . . . .	26
4.8 Evaluating the classifier . . . . .	28
<b>5 Apparent questions</b>	<b>32</b>
5.1 Scaling the network size . . . . .	32
5.2 Simplicity and invertibility . . . . .	37
<b>6 Normalization</b>	<b>42</b>
6.1 Introudicing normalization for autoencoder . . . . .	42
6.2 Using this normalization . . . . .	45
<b>7 Mixed networks</b>	<b>51</b>
7.1 Oneoff networks . . . . .	51
7.2 Latent space oneoff learning . . . . .	53
7.3 A final classifier . . . . .	54
7.4 Scaling with oneoff networks . . . . .	57
<b>8 Applying this model to other datasets</b>	<b>61</b>
8.1 Ligth dark matter . . . . .	61
8.2 Other datasets . . . . .	64
8.3 Cross comparisons . . . . .	67
<b>9 Conclusion</b>	<b>69</b>
9.1 Outlook . . . . .	70
9.2 Acknowledgements . . . . .	71
<b>Appendices</b>	<b>72</b>

<b>A</b>	<b>Understanding certain choices</b>	<b>72</b>
A.1	Changing the input feature space . . . . .	72
A.2	Is it a good idea to relearn the graph at each step? . . . . .	73
A.3	The consequences of sorting outputs by lpt . . . . .	74
A.4	The usage of a batchNormalization layer in the middle of the graph autoencoder	75
A.5	Changing the definition of the transverse momentum input . . . . .	77
A.6	Comparing our graph update layer to particleNet . . . . .	77
<b>B</b>	<b>Experiments using graph autoencoder</b>	<b>78</b>
B.1	Variating the compression size . . . . .	78
B.2	Things we learned from implementing a Graph Autoencoder in tensorflow and keras . . . . .	79
B.3	Metrik analysis . . . . .	80
B.4	How topK works exactly . . . . .	81
B.5	Trainingsize, and why graph autoencoder don't care about it . . . . .	84
B.6	Why autoencoder reproduce mean values . . . . .	85
<b>C</b>	<b>Overview of less useful networks</b>	<b>87</b>
C.1	Failed approaches . . . . .	87
C.2	The first graph autoencoder that could be considered working . . . . .	88
C.3	Improving autoencoder . . . . .	90
C.4	Improving autoencoder even further? . . . . .	93
C.5	The compression algorithm that we wish we would be able to write . . . . .	96
<b>D</b>	<b>More problems while writing a graph autoencoder</b>	<b>98</b>
D.1	Choosing the righth compression size . . . . .	98
D.2	Building identities out of graphs . . . . .	98
D.3	Is permutation invariance good or bad? . . . . .	100
D.4	Why use graph autoencoder . . . . .	100
D.5	Why not to use graph autoencoder . . . . .	100
<b>E</b>	<b>Understanding Oneoff networks with more precision</b>	<b>106</b>
E.1	Other algorithms . . . . .	106
E.2	Different algorithms for latent space training . . . . .	108
E.3	Oneoff math . . . . .	109
E.4	Self improving oneoff networks . . . . .	111
E.5	How an oneoff network can become noninvertible . . . . .	115
E.6	Why c addition might not be perfect . . . . .	117
<b>F</b>	<b>Other usecases for grapa</b>	<b>118</b>
F.1	Abnormal account detection for social networks . . . . .	118
F.2	Accelarating molecular networks through pooling . . . . .	121
F.3	High level machine learning and feynman diagramms . . . . .	122
F.4	Graph like generators and onoff initializers . . . . .	127
<b>G</b>	<b>Additional Figures</b>	<b>129</b>
	<b>List of Figures</b>	<b>133</b>
	<b>List of Tables</b>	<b>137</b>
	<b>References</b>	<b>137</b>

# 1 Motivation

After starting currently the biggest particle accelerator (The Large Hadron Collider, in short LHC) in 2008, hopes were high that it would allow us to find many signs of new physics[22]. Now, more than a decade later, the only notable discovery was the measurement of the last standard model particle with the detection of the higgs boson in 2012 [5]. And even though this is a remarkable achievement, it seems a bit unsatisfactory: There seems to be nothing wrong with the standard model on a particle physics level, as we have no clear measurements violating it. But we still know that it has to be incomplete, as it does not provide any explanation for the nature of dark matter.

To combat this, there is a growing trend of improving analysis tools. One of these tools is machine learning. Machine learning, and its subset deep learning, allow you to learn specific tasks from data. And even though the LHC did not find any irregularities yet, it still generated petabytes of data which can be used for machine learning. Next to other applications, this is currently used at the LHC for jet classification and track reconstruction ([23] Gives a more complete overview). One thing that makes applying machine learning to particle physics so attractive, is that they are able to work on very low level data. While classical jet classification algorithms use high level jet observables, like for example its invariant mass, machine learning is capable of using detector level information to extract their own features. This can make features from the jet substructure accessible to a classifier and is interesting since this ability of machine learning algorithms to work on low level data might allow them also to find anomalies that were previously inaccessible. To do this, jet physics seems to be a prime example: Raw jets are fairly hard to understand for a human, while being easily presentable to a machine learning algorithm. And also certain BSM (Beyond Standard Model) models, explaining for example dark matter, should be detectable by jet physics[3].

In recent times, a special kind of deep learning, graph neuronal networks, have become very interesting. While most neuronal networks look at pictures of energy depositions similar to those of a calorimeter, graph neuronal networks look at the actual measured 4 momenta and create a graph of particles. This makes it possible to encode relations between particles in a machine learning algorithm. This results in graph neuronal networks being very useful for jet classification [48].

But next to simply improving the quality of a classifier, there is another task that should be considered: These neuronal networks are trained supervised, meaning that they need to have examples for every anomalous event. This also means that a neuronal network trained supervised can only find specific models. And since even only considering dark matter, there are thousands of suggested models, this makes testing every alternative an exhausting job. So nowadays there is a growing trend of training neuronal networks unsupervisedly: Using a special kind of neuronal network, called an autoencoder, you can let your model learn the specifics of a set of datapoints which are considered normal. This allows those neuronal networks to find jets that dont match this definition of normality [24]. It would allow them to find anything that does not match the current understanding of jet physics, without needing to know anything more.

Graph neuronal networks have not yet been applied to this unsupervised task. And since apparently they work well for the supervised task, the following chapters will apply them to the unsupervised task: After introducing some basic concepts in chapters 2 and 3, chapter 4 Focuses on the technical implementation of this unsupervised algorithm. Since this implementation is not trivial, we provide our resulting code in an easily accessible way at <https://grapa.readthedocs.io/en/latest/> . You also find a digital version of this thesis there. Afterwards chapter 5 highlights some physical problems with the resulting anomaly detection algorithm. These are then solved in chapters 6 and 7, creating a much more useful anomaly

detection algorithm. This algorithm is then used in chapter 8 on other datasets, showing that it is able to find much more general new physics than the algorithm implemented in chapter 4. Finally chapter 9 provides a conclusion and some final thoughts on how to improve our algorithm further, which is followed by our extensive appendix going into detail about the most interesting parts of this thesis.

## 2 Introduction and literature

Referenced in: [1]

### 2.1 New physics

Modern particle physics seems to be in a standstill: The standard model seems to explain everything on a small size, while also being clearly incomplete. At the same time, none of the suggested extensions seems work, which is why there are now approaches changing the fundamental way we do science. One of this approaches is suggested by QCDorWhat [24] Instead of finding new physics events by hypothesizing theories and testing them afterwards, they use an anomaly detection algorithm to filter out events that don't match your expectation. This would allow you to find events representing new physics models without needing to suggest these models first. They work in jet physics, trying to find anomalous jets that are generated by the decay of a top quark, while only knowing about those jets, which are generated by the parton shower of QCD<sup>1</sup> Particles with lower mass. You can think of this task as trying to finding new physics, while only knowing as much as we did before the detection of the top quark in 1995[14]. So when we would be able to find top quarks at this point in time, we might also be able to apply such an algorithm to the LHC now and use it to understand physics no human knows about yet, which is why we also test most of our algorithm on this task.

### 2.2 Neuronal networks and autoencoder

Neural networks can be understood as being able to learn to reproduce any function you train them on. The anomaly detection algorithm QCDorWhat uses, is based on a special kind of neuronal network, which is called an Autoencoder [34]. Here this function learned is the identity and thus should simply reproduce everything you feed into the autoencoder. Since learning an identity is usually trivial, a point of lower dimension in the middle of the network is introduced. This compressed (latent) State, containing less information than the network input, is generated by a learnable function called encoder. After the encoder, the latent space serves as input to another learnable function(called the decoder). This functions reconstructs the input again from the compressed state. Since the reduced dimension requires the network to throw away some information, this results in a reconstruction which is usually not perfect. It can still be quite good, as long as the data contains some patterns. Learning those patterns allows the autoencoder to work as anomaly detector, as data with different patterns cannot be reproduced.

---

<sup>1</sup>Quantum ChromoDynamics, with QCD particles you usually mean low mass quarks and gluons.

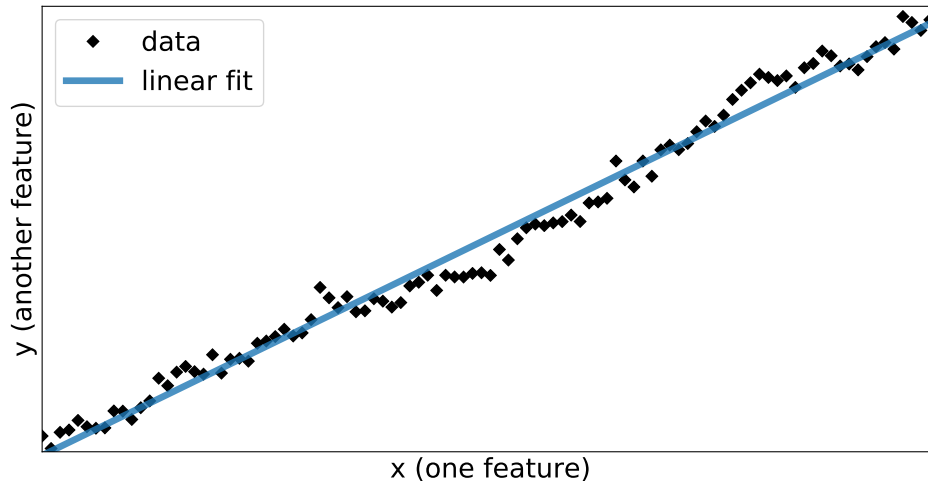


Figure 2.1: A simple example on how an autoencoder can reduce the number of parameters that is needed to approximately encode an event. Instead of using two variables  $x$  and  $y$  to define each of the points, you can use only the  $x$  value as latent space and an approximation of the  $y$  value as a function of this latent space  $x$  value

As seen in figure 2.1, to completely encode the data you would still require 2 dimensions (a  $x$  and a  $y$  Value), but you can approximately encode them into 1 dimension quite well. You do this by using one value as this compressed state and reconstructing the second one in the decoder, as a linear function of the compressed state<sup>2 3</sup>.

This combination of a compressor and a decompressor can be quite useful in multiple ways. Ignoring the obvious task of compressing data (see [53] for an example of an autoencoder used for this in particle physics), you can give the decompressor noise to generate new versions of an already known kind of data (see [40]), and even though nowadays GANs(Generative adversarial networks see [40] or F.4) Are used for this task, autoencoder have still some benefits, by allowing for more control over the generated data. This works, since (for good autoencoders<sup>4</sup>) similarity in the compressed space represent similarity of the inputs. This does mean, that by identifying features in the input space you can change just one specific attribute of the input, and you can also use this to combine the features of two inputs into one, see for example [10] and figure 2.2.

<sup>2</sup>Since the number of trainingsamples is finite, you could map every sample into an index, and map those indices again onto the inputs. This would reach a zero loss for any input with an compression size of 1. The problem is not only that is finding such a function quite hard for a neural network, it would also not be useful at all: On any new data (for example the validation data), the network would not work at all. This is why these kind of functions are a part of what you can call overfitting for an autoencoder.

<sup>3</sup>In practice, this data contains structural noise, which is why the autoencoder would not learn a linear function, but a more complicated one, better representing the data.

<sup>4</sup>This works better in a special way of training an autoencoder, called a variational autoencoder [29].

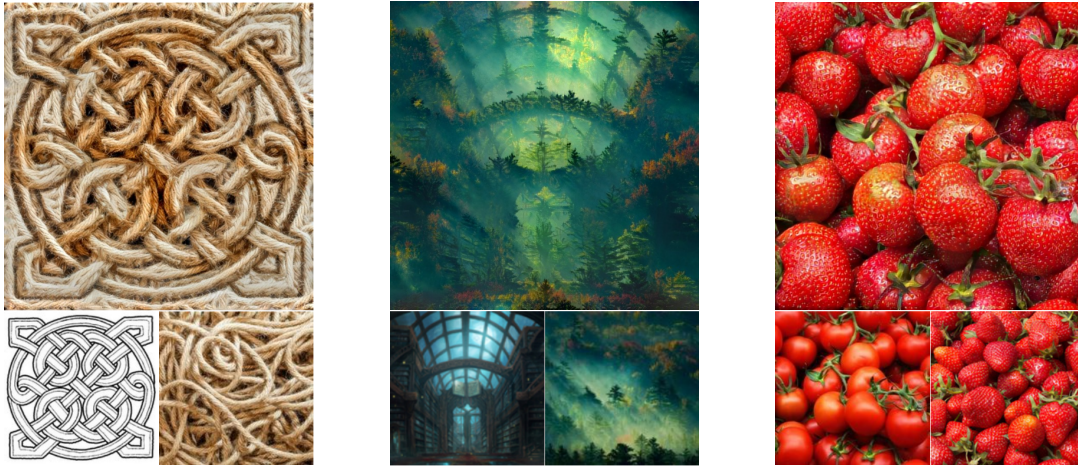


Figure 2.2: Example images showing how an autoencoder can combine two images into one. Taken from [1], generated by Ember, Bruno and ArgonOl

The application of autoencoders that is focussed on in this work, is the detection of anomalies. Introduced on the informatics side by [52] and used in particle physics for example in [43]: A well trained autoencoder should only be able to reconstruct the features it is trained on well. This means that you can use the reconstruction loss<sup>5</sup> of this autoencoder, to find events that are not of the same type as the data the autoencoder is trained on. This allows QCDorWhat [24] to find top jets in a background of QCD jets with a notable precision, without ever needing to know how a top jet looks like.

The task of anomaly detection, as finding abnormal events, has a surprising amount of use cases: From improving the purity of a dataset [7] to fraud and fault detection (see [19] and [45] respectively). To achieve this, there are also many more algorithms than just autoencoders. (These will be introduced and applied in appendix E.2). One thing most of them share with autoencoder is their unsupervised training. In contrast to usual neural networks, they never have to see examples of anomalous data, which gives them applications where there is no anomalous data jet, for example in motor failure detection and nuclear safety (see [42] and [51]). These are not only tasks for which generating anomalous data is nearly impossible, but also applications that need to find anomalies with very high accuracy: This is very similar to the the task of finding new physics (of arbitrary form) in an abundance of noise.

---

<sup>5</sup>The difference between input and output of the autoencoder, measured in a way discussed in chapter 4.5.

## 2.3 Graphs

Referenced in: [2.4] [A.3]

A graph[26] is a mathematical concept, that allows you to represent a more general form of data than those encoded in vectors. Most importantly, graphs allow for storing relational information of an arbitrary<sup>6</sup> amount of objects. This is done, by defining two objects: Nodes which are the objects of interest and can be mathematically described by vectors<sup>7</sup> and edges that are pairs of connected nodes and thus encode the relation between your objects of interest<sup>8</sup>. See 2.3 for a simple example showing how a graph can for example encode features of a city map.

---

<sup>6</sup>For computational reasons, graphs are not completely unbounded in the following chapters, but have a maximum size up to which their size is arbitrary.

<sup>7</sup>In theory you would not need to be able to define those objects as only vectors, but for practical application this is quite useful. The more complicated compression algorithm, which is described in Appendix C.4 could be interpreted as using graphs themselves as the information encoded in those nodes.

<sup>8</sup>There are multiple extensions for this simple graph, the two most important ones are directed graphs, in which the edges gain a direction, and thus a connection between node  $i$  and  $j$  does not automatically imply a connection between  $j$  and  $i$  and also weighted graphs, in which each edge gains an additional value, that encodes how strong the connection between two nodes is.

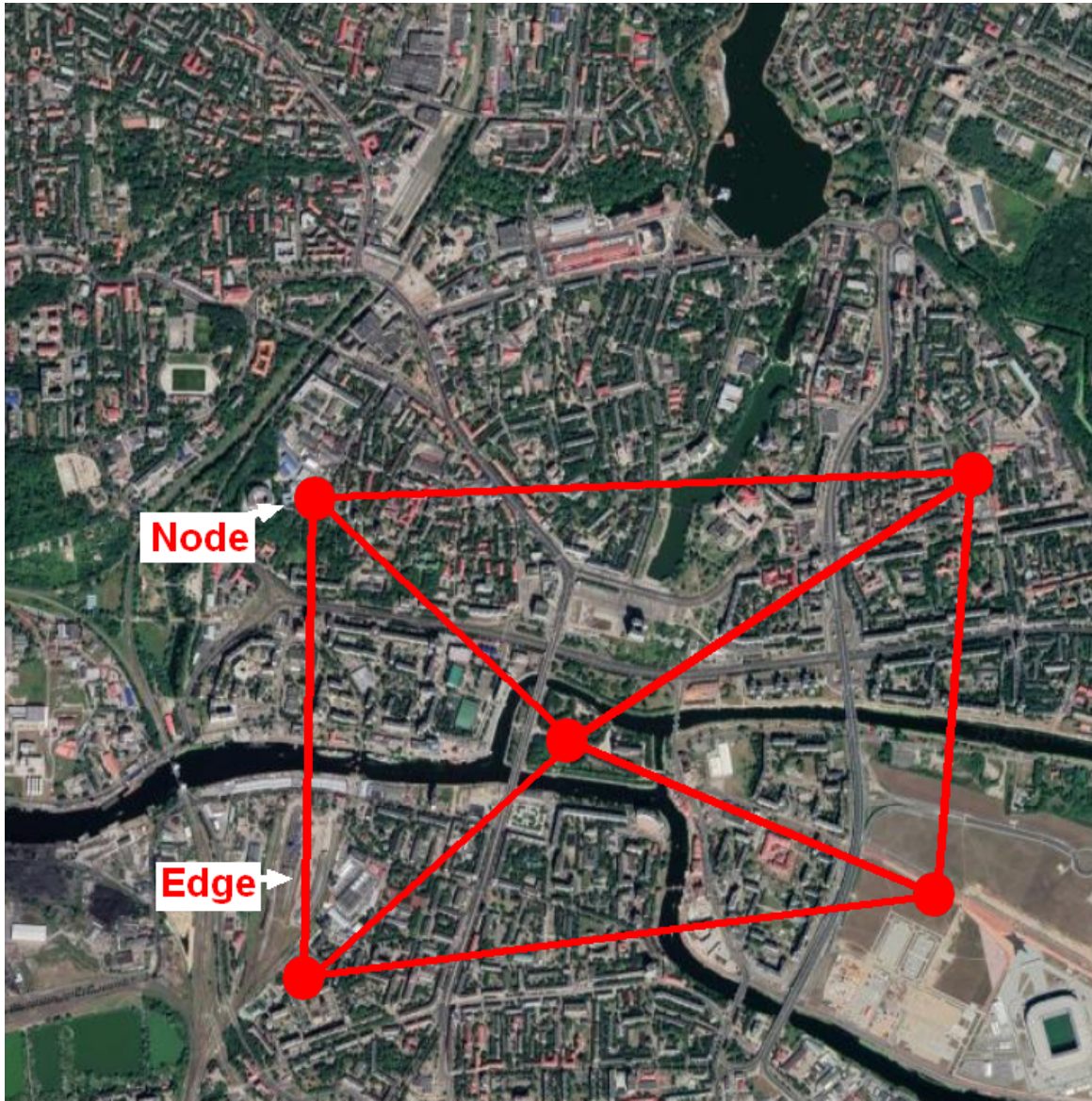


Figure 2.3: A representation of the city regions as a graph: You can understand a map as a graph, where each region becomes a node and bridges between them represent edges. Here using a map from [2]

Mathematically, these nodes and relations are defined in a list of feature vectors<sup>9</sup>  $X$  that stores the features of each node, and an adjacency matrix  $A$ , which components  $A_i^j$  are 1, if the nodes  $i$  and  $j$  are connected, and 0 if not. This graph is usually invariant under permutation of the node indices. You achieve this by permuting<sup>10</sup> the adjacency matrix in the same way the feature vectors are permuted<sup>11</sup>, and by requiring any action on the graph to be permutation invariant. This action is usually also local, and thus only acts on each node and the mean<sup>12</sup> of nodes that are connected to the current node. This has the benefit of making graphs ideal for

<sup>9</sup>Technically this is equivalent to a matrix, but list of vectors is more intuitive.

<sup>10</sup>With permuting we here mean switching two indices, or more generally multiplying with a permutation matrix.

<sup>11</sup>This is the reason why we don't call the list of feature vectors a matrix: As a matrix permutation requires permutation matrices on each side ( $p \cdot A \cdot p$ ), the feature vector "Matrix" only requires one permutation matrix ( $X \cdot p$ ).

<sup>12</sup>You also need to require each local action to be symmetric under changing the input ordering, since else the output can depend on the order of the nodes. The usual way that is achieved is by using a function like the mean on (or the maximum value of) all neighbouring nodes.

modeling interactions between high numbers of objects, as the functions don't change as you add more nodes to the graph. In informatics, this is for example useful for social networks[15]: Data that consists out of a huge amount of nodes, in which mostly only connected nodes (friends) affect each other, are perfect applications for graphs, since else you would need to update your model every time a new user joins (see for an example appendix F.1). In physics, this reminds of nuclear science, and the approximation of pair interaction potentials[18], and so there are applications using this kind of molecule encoding for chemical feature extraction (for a simple example look at appendix F.2) [17] and medicine [46]. Next to those relational applications, there are also applications that are not utilizing an existing relation, but use the locality of the graph structure to encode the similarity of given data. This is done by letting the sense of similarity between nodes be a learnable function. For example, by using a topK algorithm (each node is connected to its nearest  $K$  neighbours, see appendix B.4), you can implement a learnable version of whatever distance means. This allows networks like for example ParticleNet [44], which uses a special kind of neural network, that is able to work on graphs, to separate top and QCD jets in a supervised way. They use the graph structure to be able to define and redefine multiple times, which detected particles (nodes), should be considered close to each other. This results in ParticleNet working quite well as a supervised classifier(see [27]).

## 2.4 Graph autoencoder

ParticleNet might be well suited for classifying jets, but when you want to use it for finding new physics, then its supervised approach is still problematic. Supervised training means, that each new physics model can only be detected, if you train a special network just for it. Not only would this need a lot of networks, with the corresponding high number of false positives, but this also limits their effectiveness, as you can only find new physics that has already been thought of before. But maybe could you use the graph structure that makes ParticleNet so great and combine it with the unsupervised approach of QCDorWhat. This is the main idea that is implemented in this thesis. What you require is a autoencoder that can utilize graphs, which is a task that is not trivial: Creating something like a graph autoencoder has some problems, for example is a compression step usually not local<sup>13</sup>. Most authors shy away from any approach that changes the graph size(see for example [31]). The first approach you find, when you search for a graph autoencoder, is paper [30] and a lot of paper referencing it. The main problem using this is, that they use only one fixed adjacency matrix, and thus one equal graph setup, for any input and at any point in the network. This allows for neither the learnable meaning of similarity that seems to make ParticleNet so good, the variable inputsize discussed in chapter 2.3 and, probably worst, not for any structural difference in different jets.

Other approaches come from the problem of graph pooling operations, meaning the definition of some kind of layer, that takes a graph as input, and returns a smaller graph as the result of some learnable function<sup>14</sup>. DiffPool [54] and mincutpool [12] might be good examples for this, but graph u nets [21] stand out, since they also give a suggestion on how to implement an anti pooling layer, and thus allows for a graph autoencoder in the way we require it here. This is the reason, why the first approach we tried is based on their approach. See for this appendix C.1.

---

<sup>13</sup>Graph pooling operations are quite common, since the output of a graph network usually has a different format than its input. The way this is usually achieved, is by applying a function (mean or max for example) to each node. ParticleNet for example uses a GlobalAveragePooling [33], so it calculates the average over the nodes for each feature. This kind of pooling works quite well, but is sadly not really applicable to autoencoders, since functions like a mean are not invertible in any way.

<sup>14</sup>This is not an entirely solved problem. If solved, it would allow for hirachical learning, similar to the use of pooling layers in convolutional networks. See appendix F.2 for an application of our algorithm as pooling layer.

## 3 Basic concepts

Referenced in: [1]

### 3.1 Binary classification

Referenced in: [4.6.4] [9.1]

Evaluating the difference between background<sup>15</sup> and signal<sup>16</sup> data has been studied a lot because of its use cases for example in medicine [49]. In general, you consider 4 fractions, the fraction of events that are of type background or signal, which are classified either as background and signal.

	positive	negative
positive	True positive	False positive
negative	False negative	True negative

Table 3.1: 4 fractions for evaluating boolean decision problems. Events are truly column and classified as row

#### 3.1.1 ROC curve

For most decision problems, these fractions are a function of some parameter. Consider the loss distribution in figure 3.1.

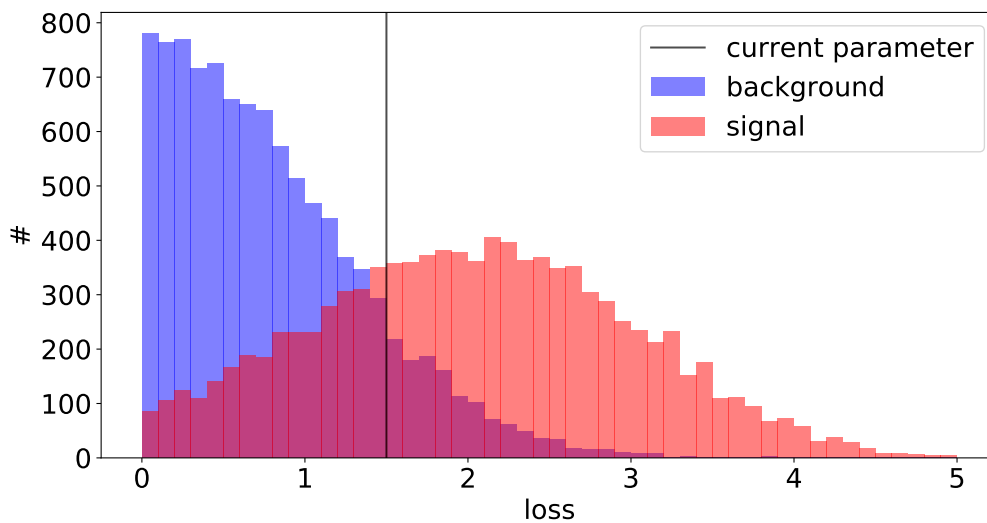


Figure 3.1: A sample loss distribution, to explain how to calculate a ROC curve. To get your ROC curve, you have to choose every possible position of the parameter. Given one parameter value, everything with a loss higher than the parameter is classified as signal, and everything with lower loss as background. Your ROC curve is now the collection of all true and false positive rates for each possible parameter value

Here this parameter is the point at which to cut the distribution in a way that everything above will be classified as signal, while also classifying everything below as background. Since the choice of this cut is completely arbitrary, we evaluate every possible parameter and plot two fractions from table 3.1 against each other:

<sup>15</sup>Here background data is the data we train our networks on. This what we consider normal.

<sup>16</sup>The signal events are those that are anomalous, and which finding is the main task.

- The first fraction, called the false positive rate, is given by the number of events that are wrongly classified as type signal divided by the number of elements that truly are background.
- The second fraction, called the true positive rate, is defined as the number of correctly classified events in the signal category, divided by the total number of signal events.

These two fractions are plotted against each other in a way showing the fraction of correctly classified events and the AUC score (see chapter 3.1.2). An example curve is shown in figure 3.2. Here the more area is under this curve the better the reconstruction is. A perfect classifier would result in this curve being 1 for every false positive rate bigger than 0.

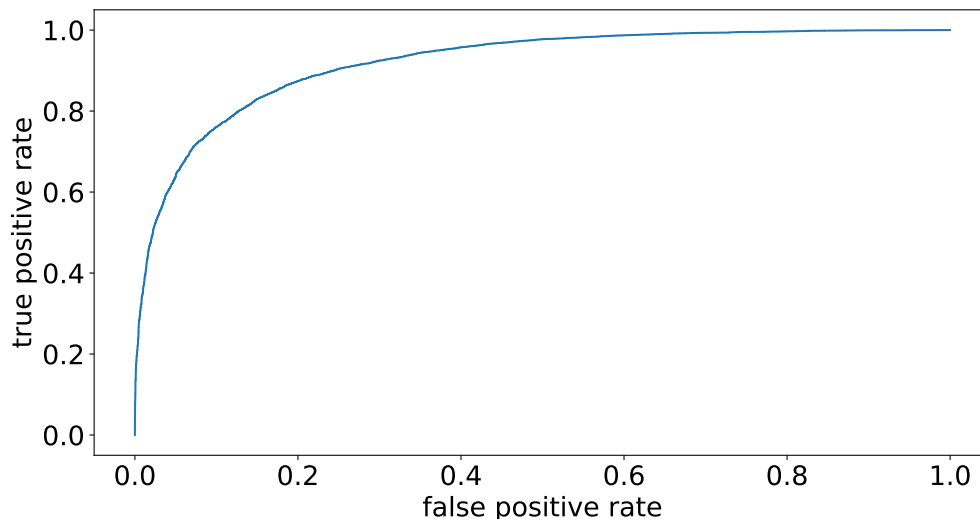


Figure 3.2: A sample ROC curve plotted in way showing its AUC score

Alternatively you visualise this curve showing the fraction of falsely called signal events, which is usually called the background rejection rate (see figure 3.3). Here again a higher curve would be better. A perfect classifier would here simply be infinite everywhere.

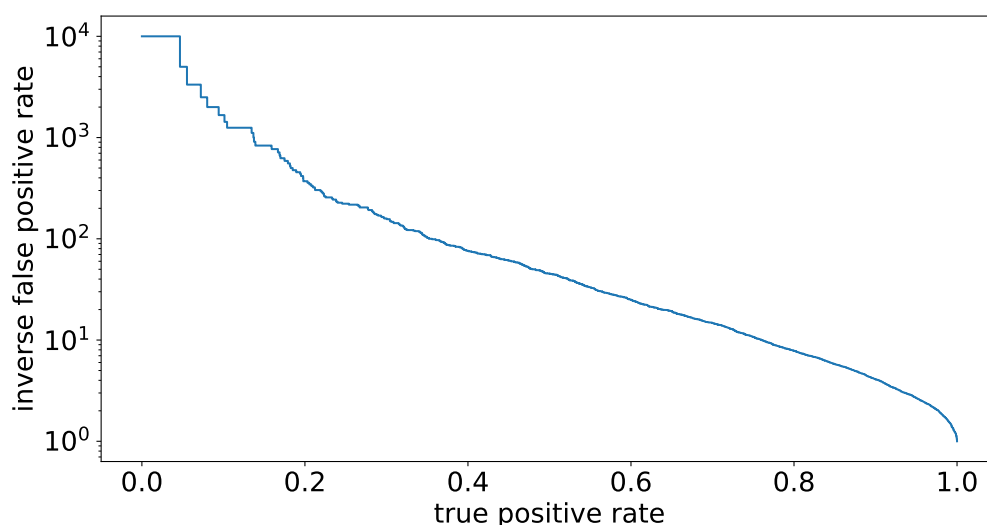


Figure 3.3: A sample ROC curve showing the background rejection rate

This resulting curve is called the ROC (Receiver operating characteristic) curve.

### 3.1.2 Area under the curve

Referenced in: [3.1.2] [4.6.4]

To simplify comparing ROC scores, you can use an AUC (Area Under the Curve) score to summarize it. This AUC score is defined as the integral of the true positive rate over the false positive rate. This simplification is not perfect, since you reduce a function into only one number, but it is fairly wide accepted, as it is much easier to interpret: A perfect score would result in an AUC score of 1, while a classifier that just guesses randomly, results in an AUC score of 0.5 and a perfect anticlassifier would result in an AUC score of 0. This still has the problem, that since not every part of the ROC curve is equally important for the current problem (if you want to test, if somebody is ill, you might prefer more false positives over more hidden illnesses), this could result in networks improving the AUC score by just changing unimportant parts of the ROC curve.

### 3.2 Datapreparation

Referenced in: [4.4.1] [4.6.4] [7.1.1] [9.1] [A.1] [C.2.2] [E.5] [F.3.3]

Here jet data provided by [28] is used. This we do, since results on this data are easier to compare. These jets have a transverse momentum between  $550 \cdot \text{GeV}$  and  $650 \cdot \text{GeV}$  and a maximum radius(  $R_i^2 = \eta_i^2 + \phi_i^2$ ) of  $R_i \leq 0.8$ .

These jets take the form of lists containing 200 momentum 4-vectors sorted by their transverse momentum. So by taking only the first  $n$  vector for each jet, you get a list of the  $n$  particles carrying the most transverse momentum, and thus probably the most important ones. If a jet is defined by less than 200 final particles(which is always the case), those remaining 4 vectors are set to 0.

In the following, every network has a fixed maximum number of particles that can be reconstructed with it. We use those particles that have the most transverse momentum, possibly including zeros. Any further preprocessing is done by the network<sup>17</sup>, namely each momentum 4 vector is transformed into a vector of 4 other variables:

- Flag: A constant 1 for each particle, but 0 if the 4 vector is 0. This input replaces the biases of our update steps, since adding a constant bias would not differentiate between those vectors that represent particles, and those that are just filler zeros. This means, that Networks using it would not be completely independent under concatting zero vectors to all inputs (increasing the graph size)<sup>18</sup>. This is definitely a minor effect, but other results appeared suggesting that it is a wise choice. We discuss them in chapter 4.4.1.
- $\Delta\eta$ :  $\eta = \log\left(\frac{p+p_3}{p-p_3}\right)/2$  (with  $p = |\vec{p}|$ ) which is shifted in such a way, that the mean of  $\Delta\eta$  is 0, since the position of the jet should not have any meaning:  $\Delta\eta = \eta - \text{mean}(\eta)$ .
- $\Delta\phi$ :  $\phi = \arctan_2(p_2, p_1)$ <sup>19</sup> which is again shifted in such a way, that the mean of  $\Delta\phi$  is 0:  $\Delta\phi = \phi - \text{mean}(\phi)$ <sup>20</sup>, since also the position of the jet in this variable should not have any meaning.
- $lp_T$ :  $p_T^2 = p_1^2 + p_2^2$ , and  $lp_T = -\log\left(\frac{p_T}{p_T^{jet}}\right)$ . We use a logarithm, to keep each value at about the same order of magnitude, which makes the training more stable. We also divide by the total jet transverse momentum, to make every jet look more similar. Finally the sign is used to keep the values positive.<sup>21</sup>

You could try to use more variables: ParticleNet for example uses 4 more variables (different representations of our variables, the energy as well as  $\Delta_R^2 = \Delta_H^2 + \Delta_\Phi^2$ . The also don't use flag), but since these variables are strongly related to other variables, this results in an autoencoder only learning those relations. This would not result in anything learned being usable as classifier. And demanding that this and more is learned, just complicates the task, without providing any real benefit<sup>22</sup>.

<sup>17</sup>The additional computation time for this is negligible compared to the graph procedures and this also allows for easier switching of data and preprocessing and even for (slightly) learnable preprocessing, like a learnable normalization.

<sup>18</sup>By adding biases to zero vectors, you get vectors that are not necessarily zero. But since the effect of a vector in the graph update step is proportional to its size (see chapter 4.1), this means, that zero vectors can have an measurable effect on non zero vectors.

<sup>19</sup>The function  $\text{atan}_2(y, x)$  is an extension of  $\arctan(y/x)$  that is able to map to the full  $2 \cdot \pi$  output space.

<sup>20</sup>Here is shifting actually not that easy to implement, since you have to consider the difference in a modular space, see appendix B.2.1 or the actual implementation at <https://github.com/psorus/grapa/blob/master/grapa/layers.py#L6488> for more information.

<sup>21</sup>A consequence is that higher transverse momenta have lower values. Since the alternative in appendix A.5 explores the effects of changing this, we don't think that this matters much.

<sup>22</sup>But see appendix A.1 for some experiments in changing the features.

### 3.3 Explaining figures used in this thesis

#### 3.3.1 Output images

Referenced in: [4.6.4] [C.3.2]

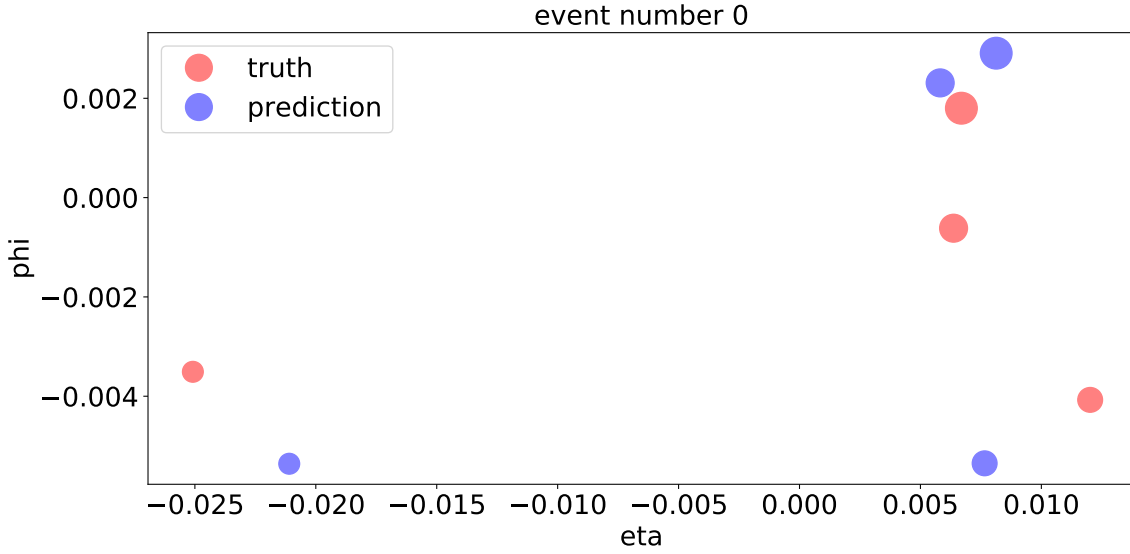


Figure 3.4: A sample reconstruction image

In figures like 3.4, you see the  $\phi$  and  $\eta$  value of each particle, including any normalization, plotted once for the input jet in red, and once for the output jet in blue. This means that a perfect network would show both jets overlapping in violet. Zero particles are not shown and there is some indication of the transverse momentum in the size of the dots, which is given proportional to<sup>23</sup>  $1 + \frac{9}{l p_T + 1}$ . This sadly does not allow you to see differences in  $l p_T$  very well, which is why we also look only at  $l p_T$  reconstruction. We show those in figures like 3.5 here as a function of the index (sorted by  $l p_T$ ).

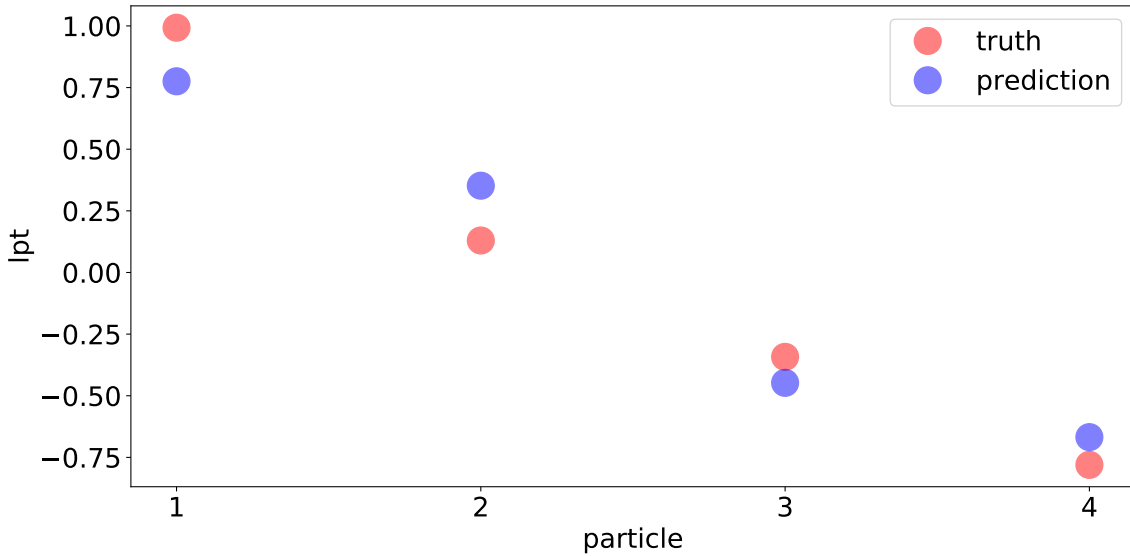


Figure 3.5: A sample momentum reconstruction image

<sup>23</sup>Inverse function, since higher  $p_T$  result in lower  $l p_T$ , and some constants to keep the radius finite.

This way of looking at the network performance is quite useful for finding patterns in the data. There seem to be networks that show a high correlation between the angles (see for example appendix C.2), and it is quite common for the reproduced values to have less spread than the input one (see appendix C.3). A problem here is, that you can only look at a finite number of images, and finding one nicely looking reproduction for pretty much each of our trained networks is not that hard. To tackle this, we always use the same event for each training set.

### 3.3.2 AUC Feature maps

Referenced in: [5.2.1]

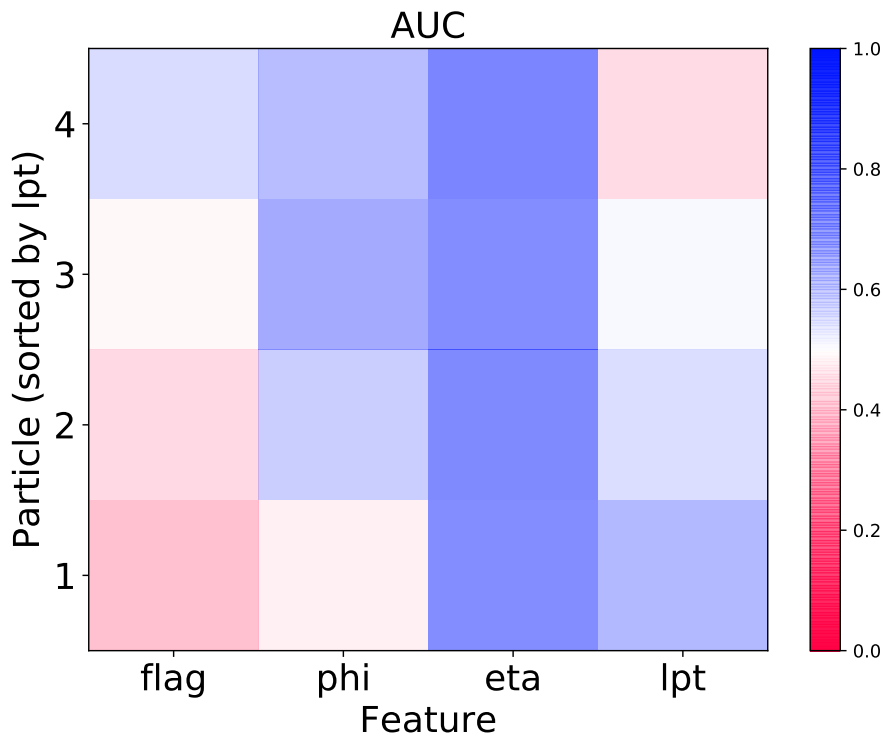


Figure 3.6: A sample AUC Featuremap

A loss is usually just a mean over a lot of losses for each feature and particle. So you could just not average them, to also be able to calculate an AUC score for each feature and particle (If the loss function described in 4.5 does not allow this, we simply use a L2 loss for this figure). These AUC scores are shown in feature maps like 3.6, showing the quality of each combination of feature on the horizontal axis and particle on the vertical axis in the form of pixels. A perfect classifier ( $AUC = 1$ ) for one pixel would result in a dark blue pixel, a perfect anti classifier ( $EQ(AUC, 0)$ ) would be represented by a dark red pixel. Finally, a useless classifier, that guesses if a jet is part of background or signal (or one that always uses the same value) ( $AUC = 1/2$ ) would be a white pixel. In short: The more colorful a pixel is, the better it is, and an autoencoder trained on QCD events should have a feature map that is blue, while an autoencoder trained on top events should be represented in red consistently.

The useful thing about those maps, is that they can show problems in the focus of the network. Since a perfect and a terrible reconstruction, both have no decision power, a network that has focus problems (meaning it reconstructs some things much better than other things, making both parts worse as classifier), can be clearly seen in those maps. Also, it is fairly common to get one feature and particle, that has alone more decision power than the whole

combined network (see 7.1 for an example and 5.1.4 for the explanation). Finally, an AUC map that is completely blue or red is quite uncommon, more probably some features are red, some are blue, which allows you to get an indication on which features are useful for the current task (see for example appendix C.2).

## 4 A working graph autoencoder

Referenced in: [1] [5.1.3] [6.1.2] [7.3.2] [C.5] [D.1] [D.5.1]

### 4.1 Graph neural networks

Referenced in: [3.2] [C.1.2] [D.2]

Graph neural networks are defined by a graph update layer. This layer takes all the feature vectors of the current graph, as well as their corresponding graph connections to return an updated feature vector. To achieve this, this layer uses two different interactions, the update step of each node itself (which is called the self interaction term here) and the update step of a node corresponding to its neighborhood in the graph (the neighbor interaction term).

Our graph update layer consists out of two matrices, a self interaction matrix, that get multiplied with each feature vector to generate the first part of a new vector, and a neighbor interaction matrix, that gets multiplied to the sum of the neighbor vectors of each node and thus forms the second part of the new vector. So written as a formula, the new vector equals (with the original feature vector  $x_i$ , the learnable self and neighbor matrices  $s_i^j$  and  $n_i^j$ , as well as the adjacency matrix  $A_i^j$  and the activation  $f$ )

$$f(n_j^k \cdot A_k^i \cdot x_i + s_j^i \cdot x_i) \quad (4.1)$$

It should be noted, that this implementation is a bit slower than the usual approach (for a reasoning on why we cannot use the more usual approach of for example ParticleNet, see appendix A.6)<sup>24</sup>, and since we don't think the implementation (see git <https://grapa.readthedocs.io/en/latest/>) is as fast possible, this is something that could be improved a lot .

#### 4.1.1 Tensorproducts

Referenced in: [4.3] [C.4.3] [D.2]

The input feature vectors  $X$  are inherently 2 dimensional. You can understand how a network updates them, by looking at how this update step would look if  $X$  would be flattened<sup>25</sup> into 1 dimension. The function used to generate this matrix from the matrices used in the update step is called a tensor product. Considering general 2x2 update matrices for the self ( $s$ ) and neighborhood ( $n$ ) interaction, as well as a given adjacency matrix  $A$  of<sup>26</sup>

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad (4.2)$$

You can calculate a 1 dimensional update formulation of the update step as  $s \otimes 1 + n \otimes a$ , which results in:

$$\begin{bmatrix} s_{00} & s_{01} & n_{00} & n_{01} & 0 & 0 \\ s_{10} & s_{11} & n_{10} & n_{11} & 0 & 0 \\ n_{00} & n_{01} & s_{00} & s_{01} & n_{00} & n_{01} \\ n_{10} & n_{11} & s_{10} & s_{11} & n_{10} & n_{11} \\ 0 & 0 & n_{00} & n_{01} & s_{00} & s_{01} \\ 0 & 0 & n_{10} & n_{11} & s_{10} & s_{11} \end{bmatrix} \quad (4.3)$$

<sup>24</sup>Especially since the usual approach can utilise GPUs better.

<sup>25</sup>A vector, which first entries are those of the first vector in  $X$ , then of those in the second vector in  $X$  and so continued until you finally have converted  $a$  vectors of size  $b$  into one vector of size  $a \cdot b$ .

<sup>26</sup>Please note that we set here the diagonal entries to zero, while in our implementation those are usually one, but this does not really matter, since this is just a change in learnable parameters. Here this is done to simplify the following calculations.

You might be able to see how a graph neural network uses less parameters than a dense implementation and understand why this is permutation invariant. Our implementation uses this exact calculation for an experiment explained in appendix D.2.

## 4.2 The compression algorithm

Referenced in: [B.4.2]

Compression is the first algorithmic problem we have to solve ourself: Find an algorithm that transforms a graph with  $n$  nodes into one with  $m \leq n$  nodes in a learnable way, without removing information<sup>27</sup>, while keeping permutation invariance<sup>28</sup> and while also being structurally invertible later on<sup>29</sup> and while being implementable in the branchless programming style of tensorflow<sup>30</sup>.

Our algorithm works as follows:

We sort each node by their last value. This last value is usually not initially given, but a learnable result of the network. After sorting each node, each set of  $c$  nodes with similar value is compressed into one output node each (using a simple dense layer<sup>31</sup>). This means, that each compression step reduces an initial number  $n$  of nodes into  $n/c$  nodes, where  $c$  has to be an integer factor of  $n$ . Also we simply ignore the edges of the graph here, since we can simply relearn them in the next stage of the network. This actually does mean, that connected nodes are more likely compressed together, as graph update steps usually average connected nodes in some way, resulting in nodes that are connected to each other being more similar, and thus in them more likely being compressed together<sup>32</sup>.

In the appendix there is a chapter giving some physical intuition about this algorithm (appendix C.4.1) and one suggesting a more complicated algorithm (appendix C.4.2) to show that more complicated algorithms might not be a great idea.

## 4.3 the decompression algorithm

Referenced in: [B.4.2]

Finding a decoding algorithm is the true challenge in writing a graph autoencoder. Luckily we wrote an encoding algorithm that can be easily inverted, and thus our decoding algorithm works just in reverse:

Define a learnable transformation (Implemented as a simple dense network) that is able to map a single node into  $c$  nodes and apply this to each node. The graph connections could be relearned again after this step, but it seems to be a good idea to use a more complicated function here, and so we use the tensorproduct introduced in chapter 4.1.1 to combine the graph before the decompression stage with a graph of  $c$  nodes (This can be seen letting each node becoming the same learnable graph). This graph is learnable, but constant with respect to the nodes we train on<sup>33</sup>. Also we use a fully connected graph before the first decompression stage (with size 1), since there is no graph yet.

<sup>27</sup>As a trivial algorithm, which just cuts away nodes would do.

<sup>28</sup>Which would not be the case, by for example applying a dense network to the collection of variables.

<sup>29</sup>This would not be kept by most graph pooling operations. Consider for example diffPool [54]: when you transform an arbitrary number of nodes into one, we would also have to implement a transformation that transforms a node into an arbitrary amount of nodes, which is not something easily done in tensorflow.

<sup>30</sup>Consider the algorithm explained in C.5, which is not implementable, or at least not in a reasonable time.

<sup>31</sup>It might be interesting to look at more complicated functions, but we usually saw worse networks, by employing more advanced functions here.

<sup>32</sup>This is especially useful, since input spaces having the same value multiple times are much easier compressed together.

<sup>33</sup>This results in noninteger graph connection values, and thus in weighted graphs.

As this handling of graphs is not very powerful, we also have a more complicated version here. This is discussed in appendix C.4.3): In contrast to the better encoding algorithm, this one might actually be worth considering, and is used in appendix F fairly commonly.

## 4.4 Our model setup

Referenced in: [6.1.2] [6.2.2] [7.3.2] [C.1.2]

After transforming our input 4 vectors as described in chapter 3.2, we sort them by their  $lp_T$  value to get our initial comparison value. This value will now be subject to a BatchNormalization [25] layer, which helps the network converge (see appendix A.4) and, after generating a graph between them (using a topK algorithm, see appendix B.4) and using them in 3 graph update stages, we apply a compression stage. This is where the two networks we setup here, one working on 4 the particles with the highest transverse momentum and one working on the first 9, show their first difference: The 4 node network simply compresses all 4 nodes into only one, while the 9 node network gets compressed by a factor 3, is followed by 3 graph update layers, just to be compressed again by a factor 3<sup>34</sup>. All compression stages add additional parameters, until the 4 node network has now 9 variables on its only node, while the 9 node network has 20 parameters in its node. This current stage is what is called the latent space, and thus the following layers are no longer part of the encoder but of the decoder. This decoder is build completely in reverse to the encoder: We start by decompressing the latent space once (or twice with 3 update steps inbetween for the 9 node network) and then we use 3 more graph update steps, cut excess parameters and sort each node by their last value (This value is  $lp_T$ . More about why this sorting is a good idea in appendix A.3). Now we have an input and output value to define the loss of our network in a way defined by chapter 4.5.

Since we trained at  $O(1)$  models, we cannot show every model setup, so we show only this, even though it is not optimal for the current task (Different models use 4 to 7 dimensions for the 4 node case, reaching slightly better classification scores). We still use this setup here, since later networks use the same, while being depending much more on the network setup than the models trained here.

For the training of every model, we use Tensorflow [6] and keras [16].

We train our networks with a learning rate of 0.003 and a batch size of 100 until the validation loss does not improve for 100 epochs<sup>35</sup> of 50000 Training jets<sup>36</sup>. We also train for at least 500 epochs.

### 4.4.1 Our choice not to use biases

Referenced in: [3.2]

Every time you could use a dense layer, we only use the multiplicative part of it, and ignore the usually learnable bias that is added to the output of this layer. This was originally so, since we would like the exact number of nodes in the network not to matter, as long as you can represent you whole yet. Then adding a constant to zero vectors, results in them not being zeros anymore, and thus influencing connected nodes. This is definitely a minor effect, but we kept it, since we can show that this improves the classification quality: An AUC value (that is generated in the following chapter 4.8) of 0.811 falls to 0.796 by adding biases (here only to the graph update layers in between). And even though this is again a minor effect, this is already quite strange: Biases are usually considered a really good idea, and so understanding

<sup>34</sup>In our tests it generally seemed to be a good idea to compress into only one node.

<sup>35</sup>This patience is quite high, but seems to be needed to keep the networks fairly reproducible later on.

<sup>36</sup>This number is about an order of magnitude less, than the data provided in 3.2 (600000), we use this low number here to save some time, as we can show that changing this size does not affect our training quality (see appendix B.5).

why networks work better without biases, directly resulted in an approach on how to solve some of our following problems. This is discussed in chapter 7.

## 4.5 Choosing the righth loss

Referenced in: [2.2] [3.3.2] [4.4.1] [4.6.4] [5.1.4] [5.1.5] [7.2] [9] [B.6] [C.3.2] [D.5.1] [F.3.3]

Creating a good classifier means letting the network focus on exactly the things you want it to care about. This focus can be influenced in two ways: The initial normalization, and the loss function. While the size of the initial variables is relatively straight forward<sup>37</sup>, choosing different loss functions makes less predictable changes, so here we will discuss some different losses and their effect on controlling the focus of your networks.

### 4.5.1 $L_2$ loss

Setting the loss function to be the quadratic difference between input and output still is generally not the worst idea:

$$loss = \text{mean}(x - f(x)) \quad (4.4)$$

(with the input  $x$  and the autoencoder  $f(x)$ )

Not only is this easy to implement and fast to compute, but it also punishes bigger differences more than multiple small differences (two acceptable reconstructions are preferred over one good and one bad one), which we generally prefer over the alternative (see subsection 4.5.2), but this also results in autoencoders that learns mean values: If you would need to choose either  $a$  or  $-a$ , an  $L_2$  normalized network that does not know the right choice, will choose 0 all the time: This is done since sometimes guessing the wrong result is punished more, than not choosing<sup>38</sup>. This results in the output of a  $L_2$  autoencoder usually having a lower width than its input (see for example image 4.6. This is what we want to solve, by looking at different losses.

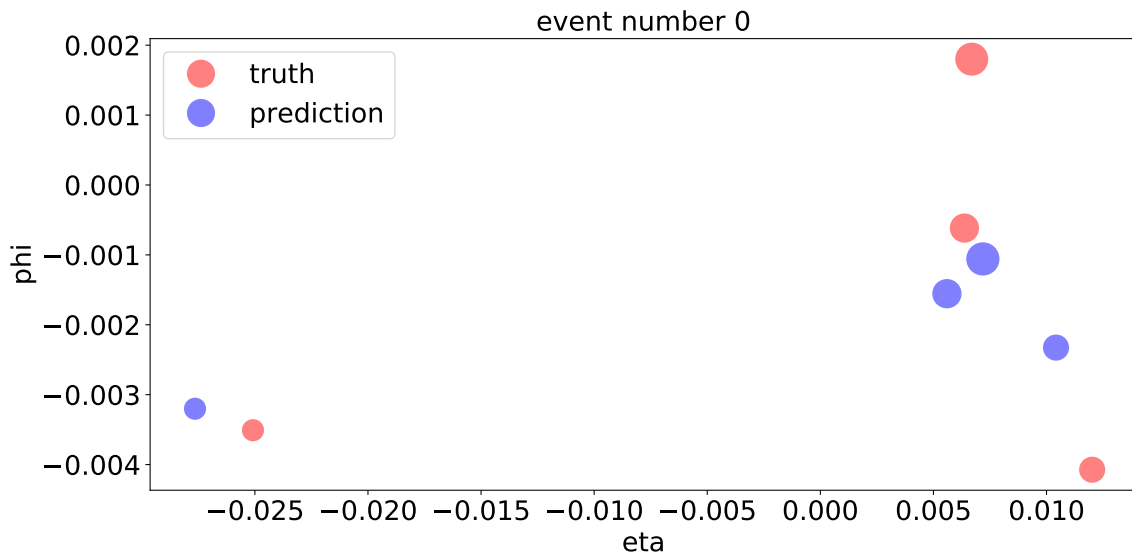


Figure 4.1: A  $L_2$  reconstruction image, the reconstructed width is often lower than the input width. Here you see this best in  $\phi$

<sup>37</sup>The bigger each value, the bigger is its contribution to the loss.

<sup>38</sup>You might ask why this is a problem, since a network that does not know anything probably should not choose one of the results, but there is a similar effect for non perfect guessing networks: Lets say the network guesses right  $\alpha$  times, then for a prediction of  $b \leq a$ , the loss is given by  $\alpha \cdot (a - b)^2 + (1 - \alpha) \cdot (a + b)^2$ , which is minimal for  $b = a \cdot (2 \cdot \alpha - 1)$ .

### 4.5.2 $L_n$ loss

Referenced in: [4.5.3]

The first other kind of loss you can look at, would be a  $L_n$  loss. For each  $2 \leq n$ , this loss still has the same problem of uncertain losses, but for smaller  $n$  it does not. L1 does not prefer lower predictions<sup>39</sup>, and a  $n$  lower than 1 would reverse the effect entirely<sup>40</sup>. This works, as those networks have a similar input width as output width, but this different loss has another effect: Since now one big loss is as bad as two small ones, Networks, that remember some values exactly, while guessing the remaining ones, are very common, as this is a much easier thing to learn.

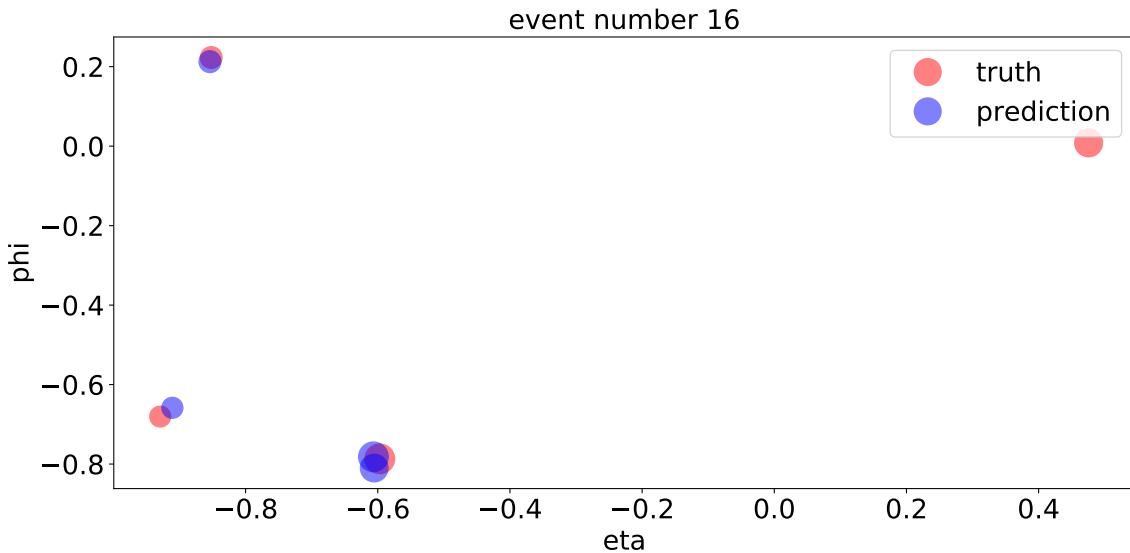


Figure 4.2: Reconstruction image for a model that only remembers 3 nodes perfectly trained with an  $L_1$  loss

Since networks like for example in 4.2, are not just setting the remaining values to some constant, but try to guess them right, we have the same situation as in the normal case: On the known data, this guessing works, and on abnormal data this works less good. But not only do less accurate guesses have way less decision power<sup>41</sup>, this also ignores the remaining values completely: Since copying is easily done even if the data is abnormal, there is no information gained here. And even if it were, this difference would be tiny compared to the loss of the guessed values, so the whole loss is dominated by only some inaccurate values, ignoring everything else. This does not mean, that  $L_n$  losses are useless, since there are networks (like image 4.3) without this problem, but retraining each network, until it does not do this, makes this loss much less desirable

<sup>39</sup>The loss described above would now look like  $a \cdot \alpha + a \cdot (1 - \alpha) - b + b = a + b \cdot (1 - 2 \cdot \alpha)$  which is minimal (for each  $0.5 \leq \alpha$  for  $b$  being as big as possible (this loss still assumes  $b \leq a$ ), and anything guessing right less than half of the time, would still result in the network learning not to guess, as we would want.

<sup>40</sup>We tried  $n = 1/2$ , but this results in NaNs (see appendix B.2.2), so we had to tweak  $\sqrt{|a - b|}$  into  $\sqrt{|a - b| + 1} - 1$  since the NaNs seem to be a result of the square root not being differentiable at 0.

<sup>41</sup>You can model this, as a fixed distance between two gaussian peaks with varying width, a plot of the relation between the width and the AUC is shown in 5.1.4.

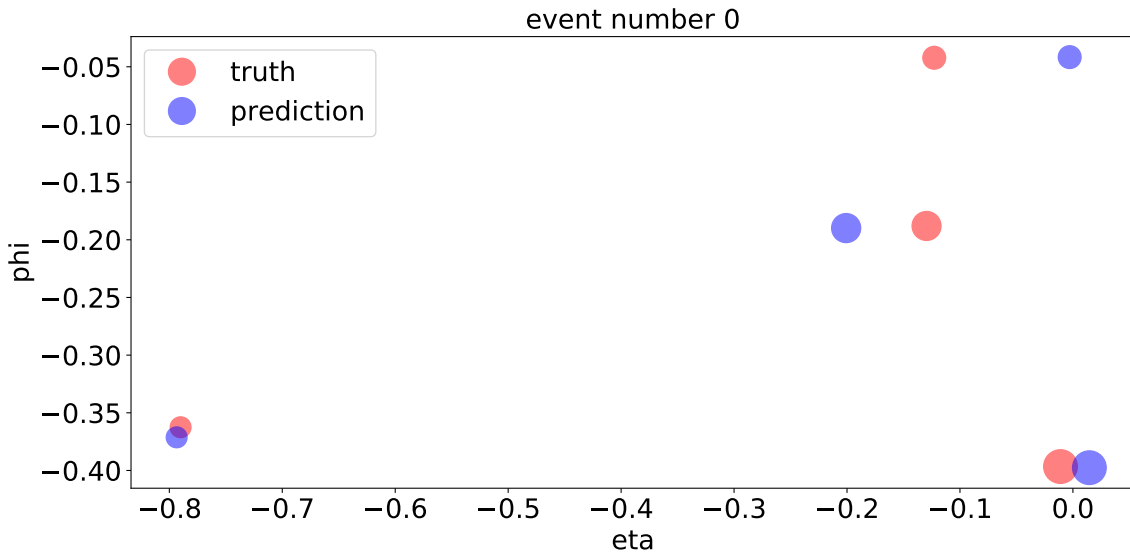


Figure 4.3: A  $L_1$  reconstruction image, not working trivially, but also not working perfectly

### 4.5.3 Image like losses

Referenced in: [9.1] [A.5]

One thing to consider, is that image like networks usually do not have these problems. The main difference in losses, is the fact, that image like networks do not compare all values to each other. Instead the only compare one value, if the other values match. A pixel is restored with zero loss, if its angles are nearly correct, and the momentum is correct. If the angles are a bit too much off, the loss is derived by comparing the input to a zero, and the newly constructed momentum to the zero of a pixel that was initially zero. This means, that each difference in angles, is punished the same (if it is bigger than some constant), and the only way, for the network to improve, is to guess those right. If we could implement this here, this would solve the problem of guessing to low, at least in angles<sup>42</sup>. To implement this, you can set the  $lp_T$  reconstruction to zero, if the angles are more than a certain difference apart. This works fine, but offers no control, is not easily differentiable, and for implementation purposes a symmetric loss function would be nice. We choose the following extension

$$(1 - f(d)) \cdot c(x) + (lp_T^a - lp_T^b) \cdot f(d) \quad (4.5)$$

where  $f(d)$  is some function of the angular difference, so for example a step function ( $f(d) = 1$  for  $d \leq d_0$  and  $f(d) = 0$  else) and  $c(x)$  is some alternative loss, for example  $c(x) = |lp_T^a + lp_T^b|$ . This extension offers a lot more flexibility and is symmetric as long as  $c(x)$  is symmetric. From our (limited) experimentation, choosing a continuous function  $f(d)$  seems to be a good idea.  $f(d) = e^{-\alpha \cdot d}$  with some  $O(\alpha) = 1$  works well. And even though this undermines the initial point, choosing  $lp_T$  as alternative loss  $c(x)$  is not such a good idea either. The point here is, that those networks heavily prefer angular information<sup>43</sup> so ideally you would choose some alternative loss, that shifts the focus more on the transverse momentum. This is not so easy, since simply choosing the difference between input and output  $lp_T$  results in the network not learning any angular information (since they now don't have any effect), but you can set  $c(x)$  to be a higher multiplicative of this difference. That being said, simply setting  $c(x) = 1$  to be a

<sup>42</sup>The problem stays the same for  $lp_T$ , since guessing wrongly zero is punished way less, than guessing wrongly  $a$ . So for the same setup as before, the new loss looks exactly the same as in the  $L_n$  case  $\alpha \cdot (a - b)^n + (1 - \alpha) \cdot (a + b)^n$ .

<sup>43</sup>This should be expected, since if  $lp_T$  is not correct, there is some loss, but if the angles are not correct, the accuracy in  $lp_T$  simply does not matter.

constant works best in practice and is what is used in the following. Another thing that is used there is a slightly different  $f(d) = \frac{k+e^{-\alpha \cdot d}}{k+1}$  with a  $k = 1/10$ . This is used here to balance the network focus more onto  $lp_T$ <sup>44</sup>. This combination works quite well and can reach reconstruction values with nearly exactly the same width for 4 particle networks

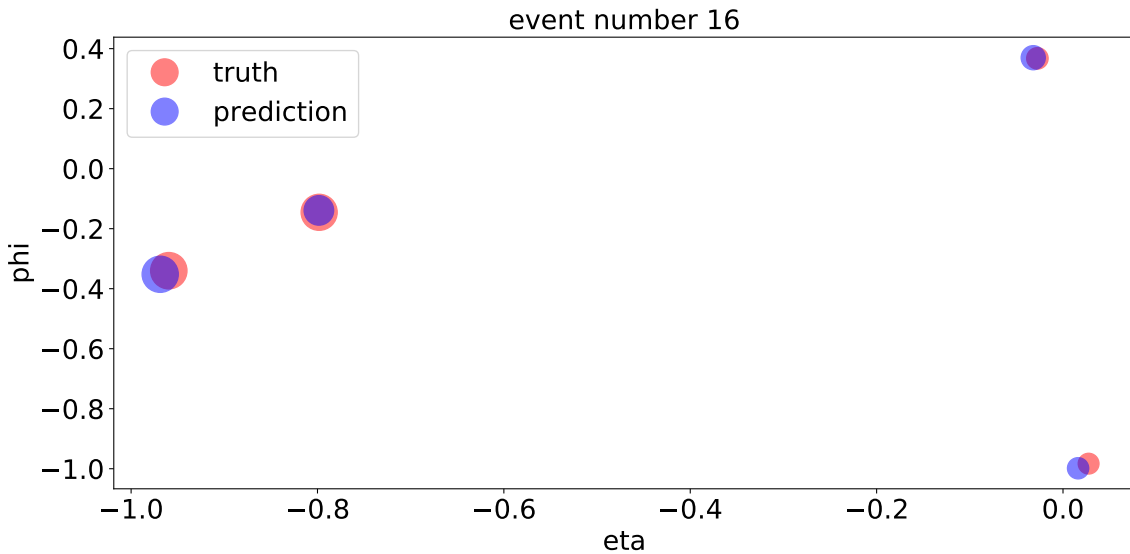


Figure 4.4: Reconstruction image of an image like loss working well

As you see in figure 4.4, this reconstruction matches the input quite well, and thus we will be using image like losses often in the following.

## 4.6 Difficulties when evaluating a model

Referenced in: [5.2.1] [8.1]

Before we can look at the results of our network, we have to look at how to judge them, as this is actually not completely trivial: We might be able to evaluate a binary classification problem (see chapter 3.1), but evaluating a network basically means doing 2 things at the same time: Creating an autoencoder and creating a classifier, so there might be situations in which the autoencoder is good, but the classifier is bad and situations in which the classifier might be good, but the autoencoder is basically useless.

### 4.6.1 AUC scores

If you want to evaluate a network, you might think that you can simply use the quality of the classifier (the AUC Score, see chapter 3.1.2) since the classifier should work by the autoencoder understanding the data, and thus should only be good if also the autoencoder is good. And in most cases this works, there is a clear relation between the quality of the autoencoder and the quality of the classifier (see chapter 6.1), but in general this is simply not true, as for example chapter 5.2.1 shows. And even if your working in a region where this relation is true, Classifier evaluation methods usually have a much higher uncertainty<sup>45</sup> than other methods, which is why in the regions in which there is a strong correlation, it was more useful to use the loss of

<sup>44</sup>This can be understood as follows: Introducing  $k$  is introducing a lower border for the effect the  $lp_T$  comparison has on one particle: Even if the distance is high,  $k$  part of the loss of this particle is still given by the momentum difference.

<sup>45</sup>Uncertainty in the sense that even a well trained network can change its AUC score by a couple of percent after retraining, even if it has the same loss.

the network to assert that the network improves, and to simply know that the AUC score will correlate.

#### 4.6.2 Losses

Using only the quality of the autoencoder and trying to optimize this would be conceptually great, as you only need to use your anomalous data once<sup>46</sup>, but this again has problems: Not only requires this still a strong relation between AUC and loss (That is here given even less, consider the problem of finding the best compression size: The loss will usually<sup>47</sup> fall by increasing the compression size, but at some point, the autoencoder can just reconstruct everything perfectly, and thus has no more classification potential), but the loss also relies heavily on the definition of the network and the normalization of the input data (see chapter 3.2), which makes comparing different networks only possible, if you neither alter the loss nor the normalization.

#### 4.6.3 Images

This cross comparison problem can be easily solved by simply looking at the reconstruction images instead of the losses<sup>48</sup>. But while this is certainly very useful, as it also allows understanding more about your network (for example, there are networks, that simply ignore some parameters, and thus have their whole loss in those parameters, this can be most easily seen by looking at the images), this still relies on the relation between AUC and loss and more importantly is less quantitative: Giving 2 images, finding out which autoencoder is better is not always an easy task, especially since what differences you might see in those images do not necessarily correspond to differences the network sees (see for this chapter 4.5). Most notably in reconstruction images, you usually care more about angular differences, while sorting by the transverse momentum introduces a slight preference for  $lp_T$  in  $L_2$  losses.

This you can solve, by also looking at the  $lp_T$  reproduction, but this demands weighting importance between images, and thus does not make evaluating images any easier.

#### 4.6.4 oneoff width

The final solution, and the solution that seems to be the best currently, is based on the things introduced in chapter 7. Because of this, it will be explained in chapter 7.1. It works, by defining the loss in a way, that does not change by changing the loss function or the initial normalization. We do this by letting the network define its own observable, which we demand to be constant over all background events. And by setting this constant to be 1, the variance of this measurable is independent under changing the inputs. This still requires some correlation between the variance of this variable and the AUC score, which we cannot assume in general, but chapter E.4 at least suggests that this is common.

---

<sup>46</sup>Usual machine learning has a problem, in which your network can learn even data that it is not trained on, simply by you comparing networks on it (this is why there is test data), the same can happen here, by you often comparing qualities of your anomalous data and since finding new test data would require you to have completely different anomalous systems, this can be difficult to do (even though we try this in chapter 8), which is why choosing to ignore your anomalies in training would be great.

<sup>47</sup>Always, except for noise and random change.

<sup>48</sup>The jet image showing input and output of the autoencoder, as explained in chapter 3.3.1.

## 4.7 Evaluating the autoencoder

Referenced in: [8] [9]

### 4.7.1 4 nodes

For 4 nodes graphs, the number of connections for each node does not really seem to matter, which is why we simply use a fully connected graph for those networks.

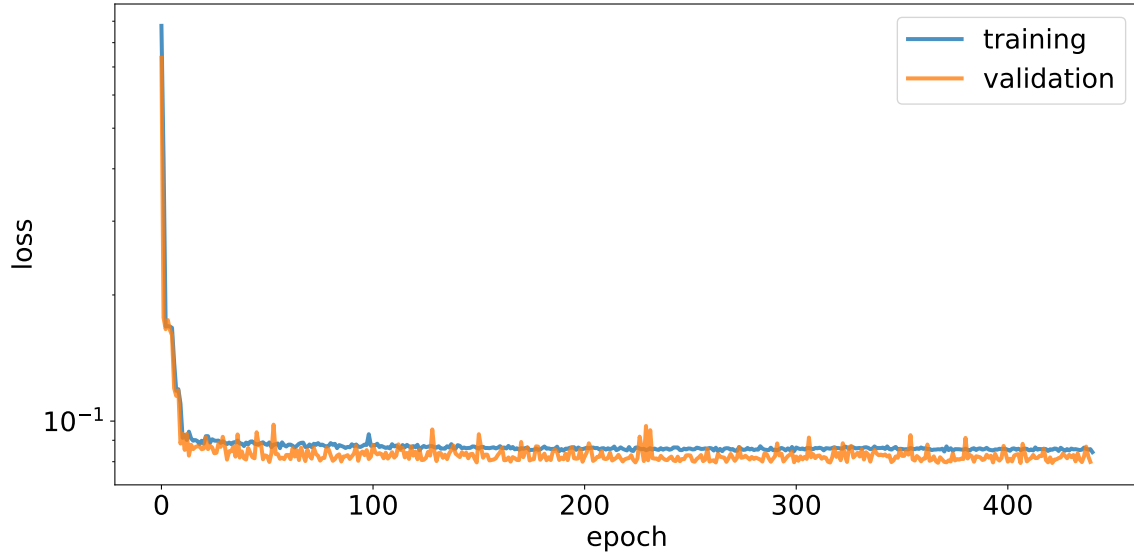


Figure 4.5: Training history for a 4 node network

In image 4.5, the training curve converges nicely to one value, and you can notice one thing here: the validation loss does not behave worse than the training loss. This is something that you would usually expect, as it is a sign of overfitting, but is something that is very common for the graph autoencoder in this thesis: It is basically impossible for those networks to overfit, in fact we can reduce the number of training values drastically without letting the network overfit (see appendix B.5).

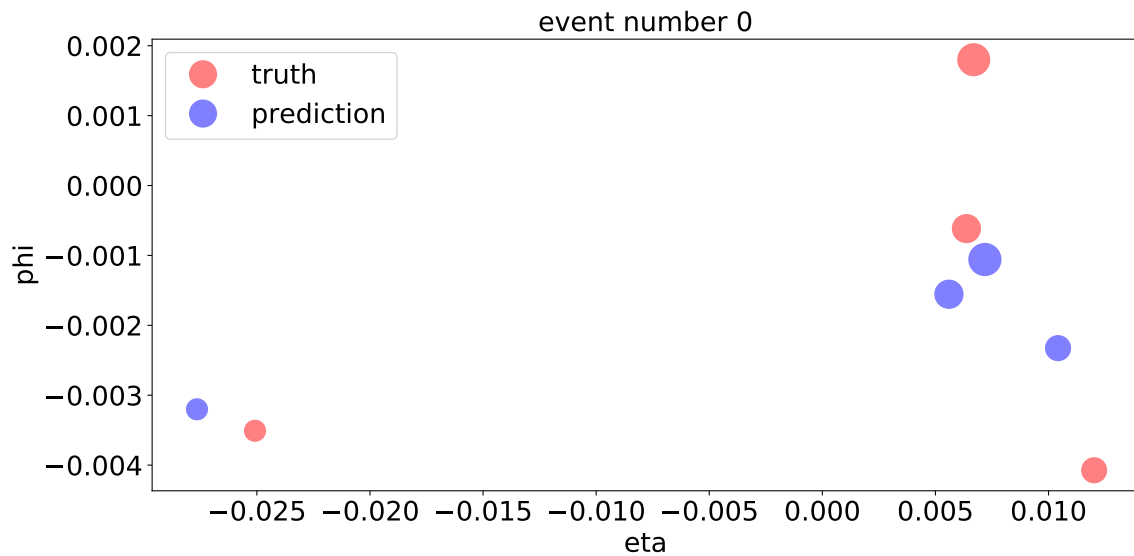


Figure 4.6: Angular reconstruction images for a 4 node image.

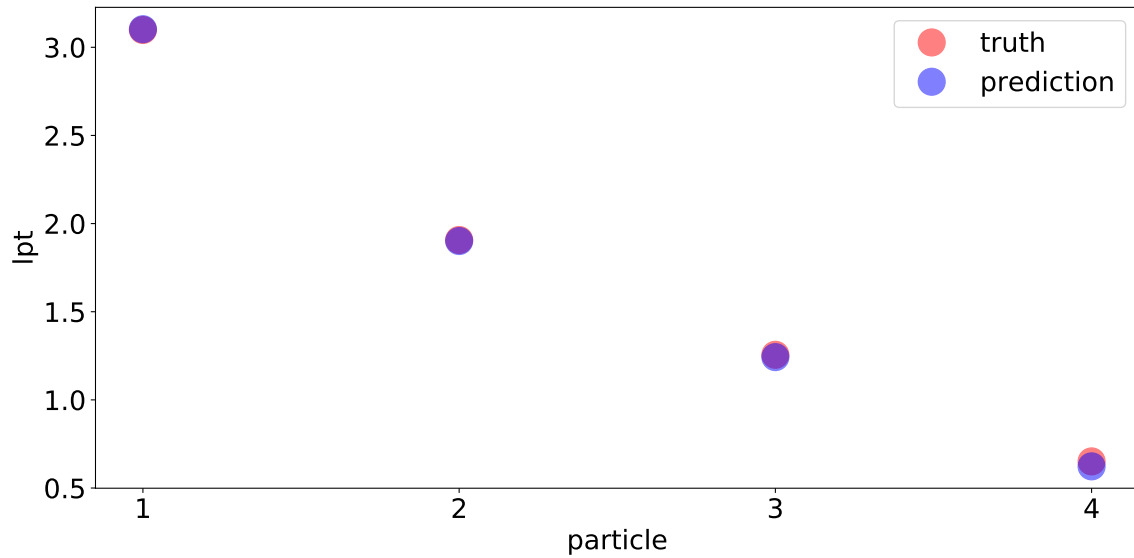


Figure 4.7: Momentum reconstruction images for a 4 node image.

The momentum reconstruction in images like 4.7 is nearly perfect and also the angular reconstruction (see 4.6) show some resemblance between the input and the output. The only problem is, that this network shown here seem to not care about  $\phi$  as much as it does about the other variables.

#### 4.7.2 9 nodes

If we increase the size of the network to 9 particles, the training fails because of NaNs (Not A Number, numerical problems). In figure 4.8 this is shown as missing validation loss values. The training stops, when the training loss is NaN, which is why this model does not train for long.

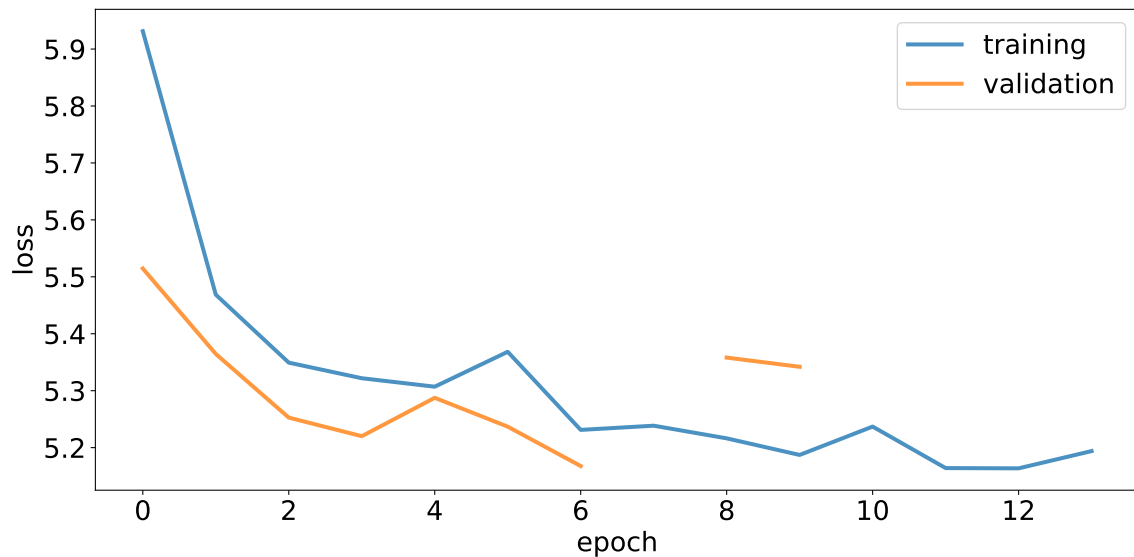


Figure 4.8: Example training history for a 9 node network showing how NaN losses hurt the training procedure

And since now this training effectively stops after only a few epochs, the loss is still really big (over 5) reconstruction is also much worse (see images 4.9 and G.1 in the appendix)

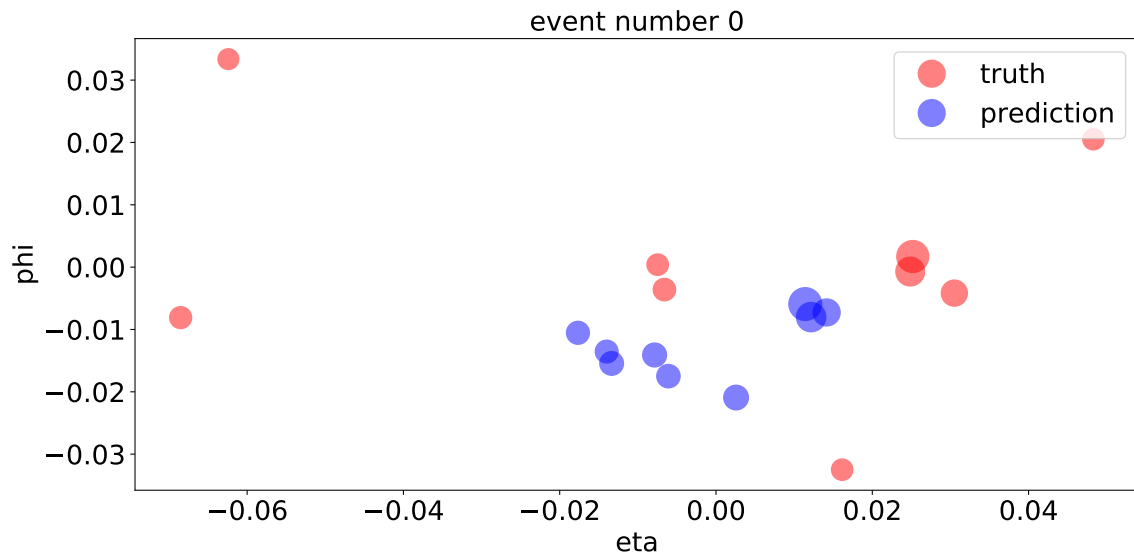


Figure 4.9: Angular reconstruction images for a 9 node image.

## 4.8 Evaluating the classifier

Referenced in: [4.4.1] [9]

### 4.8.1 4 nodes

As you see in image 4.10, these 4 particle networks already separate QCD from top jets quite well, reaching an AUC score of over 0.81 in 4.11, which is quite good considering we only use 4 particles. By changing this networks parameters you can even reach AUC's upwards of 0.85.

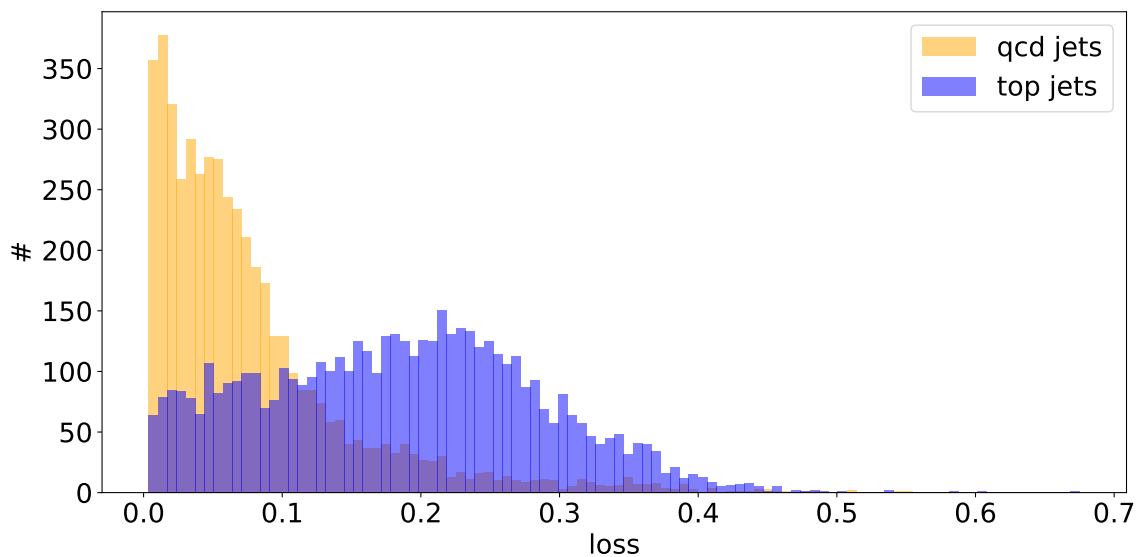


Figure 4.10: Loss distribution of our 4 particle network

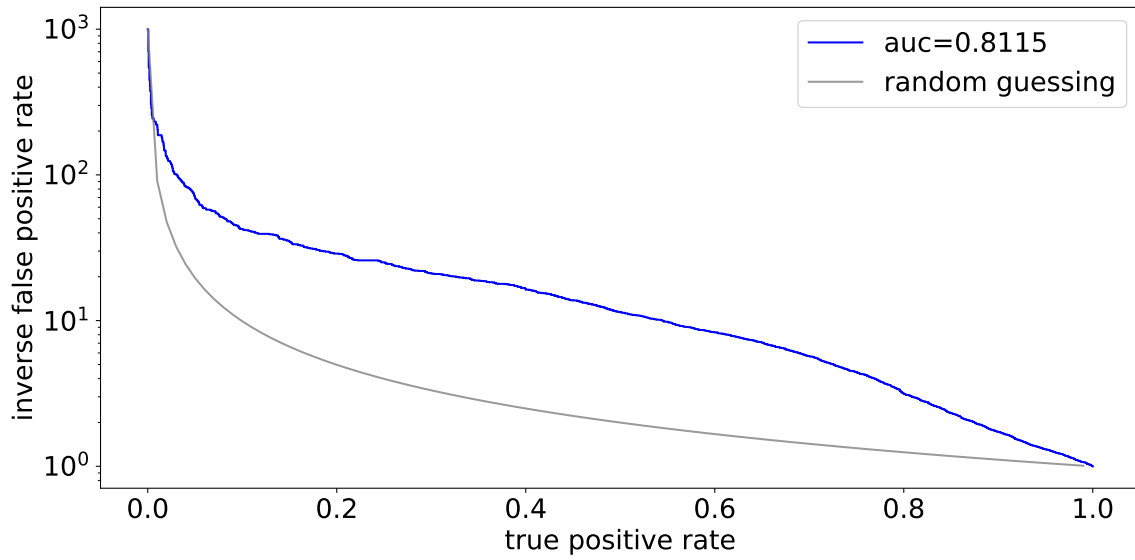


Figure 4.11: Roc curve for our 4 particle network

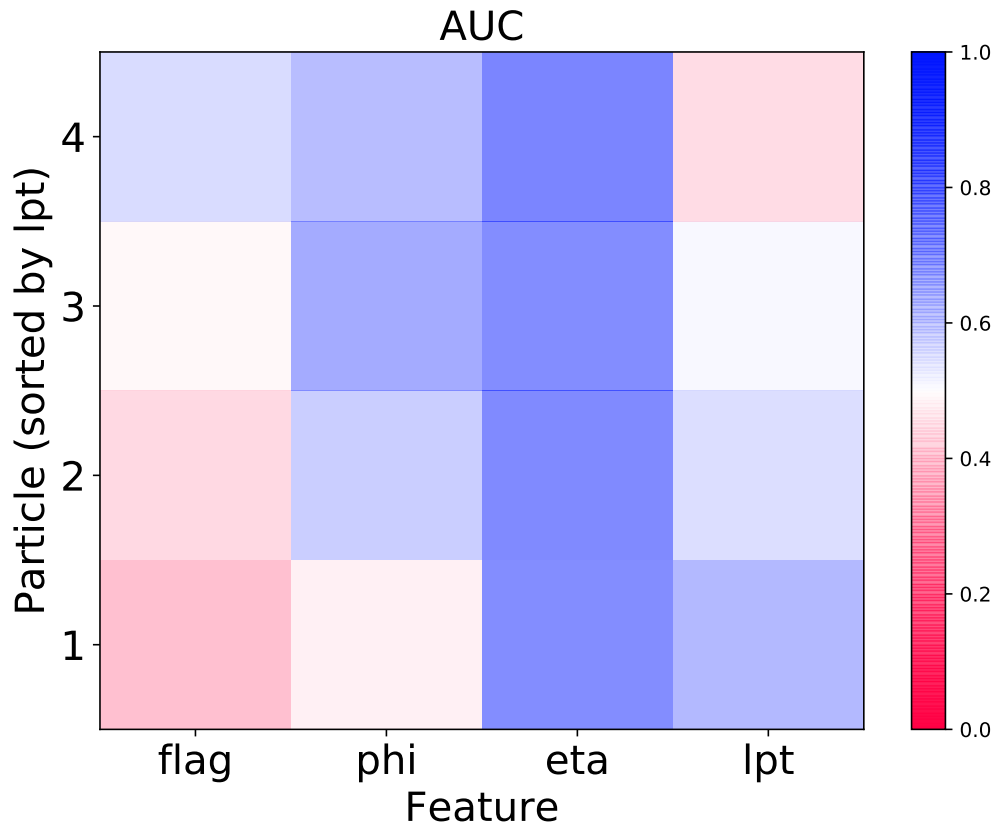


Figure 4.12: AUC feature map for 4 nodes.

Figure 5.6 shows, that this good AUC score is mostly a product of the angular parts of the loss function, as only using them reaches already an AUC value of 0.78

#### 4.8.2 9 nodes

Since the 9 node network does not work well as an autoencoder, we don't expect it to work well as a classifier.

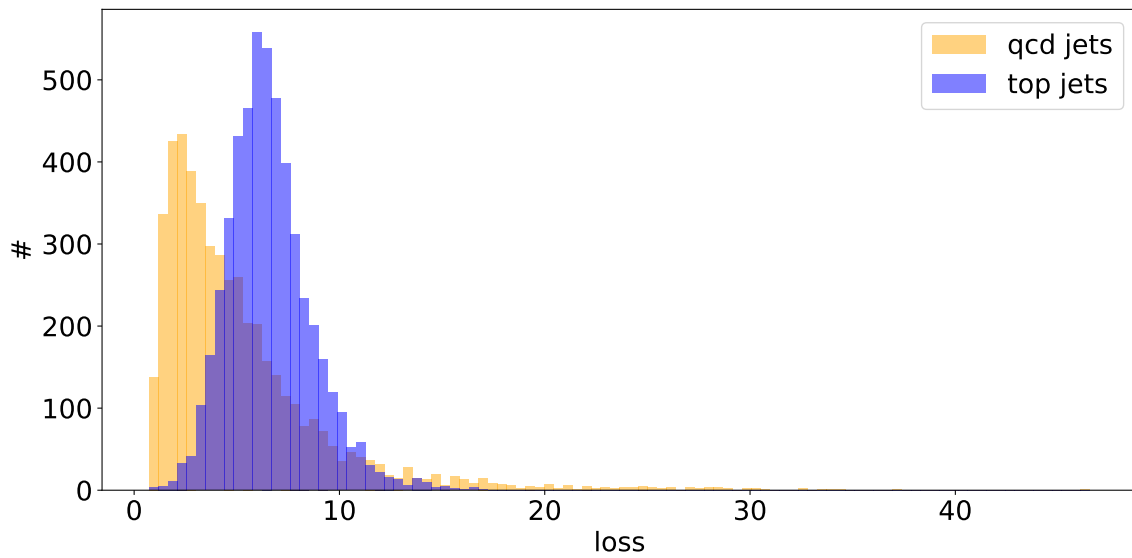


Figure 4.13: Loss distribution for the 9 particle network

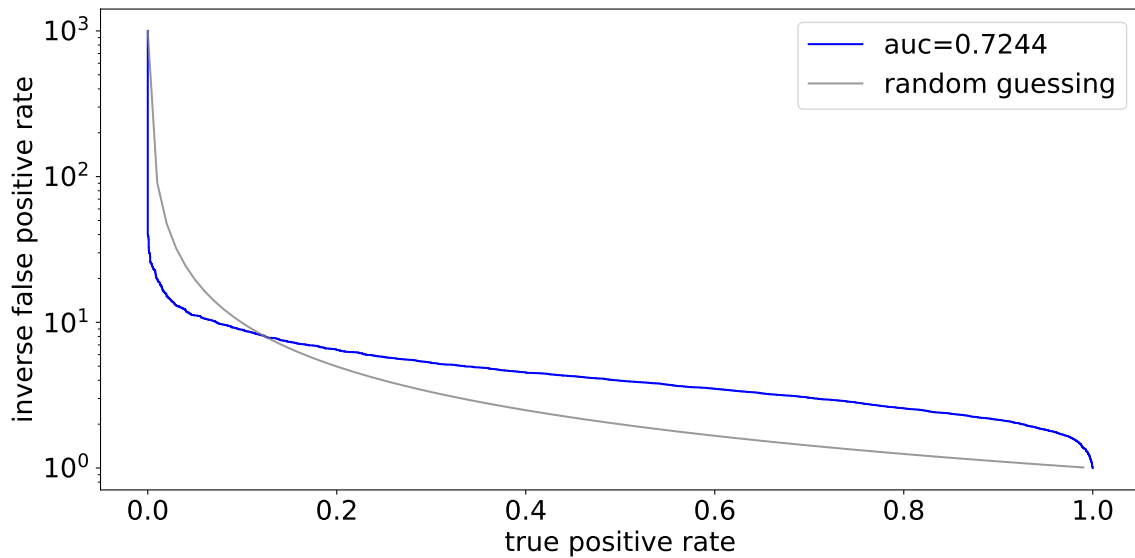


Figure 4.14: Roc curve for our 9 particle network

In figures 4.13 and 4.14 you get a worse AUC score of a bit over 0.72. This is still an acceptable classification, while the reconstruction is terrible. This inconsistency will be addressed in the next chapter 5.1 and solved in chapter 7.4.2.

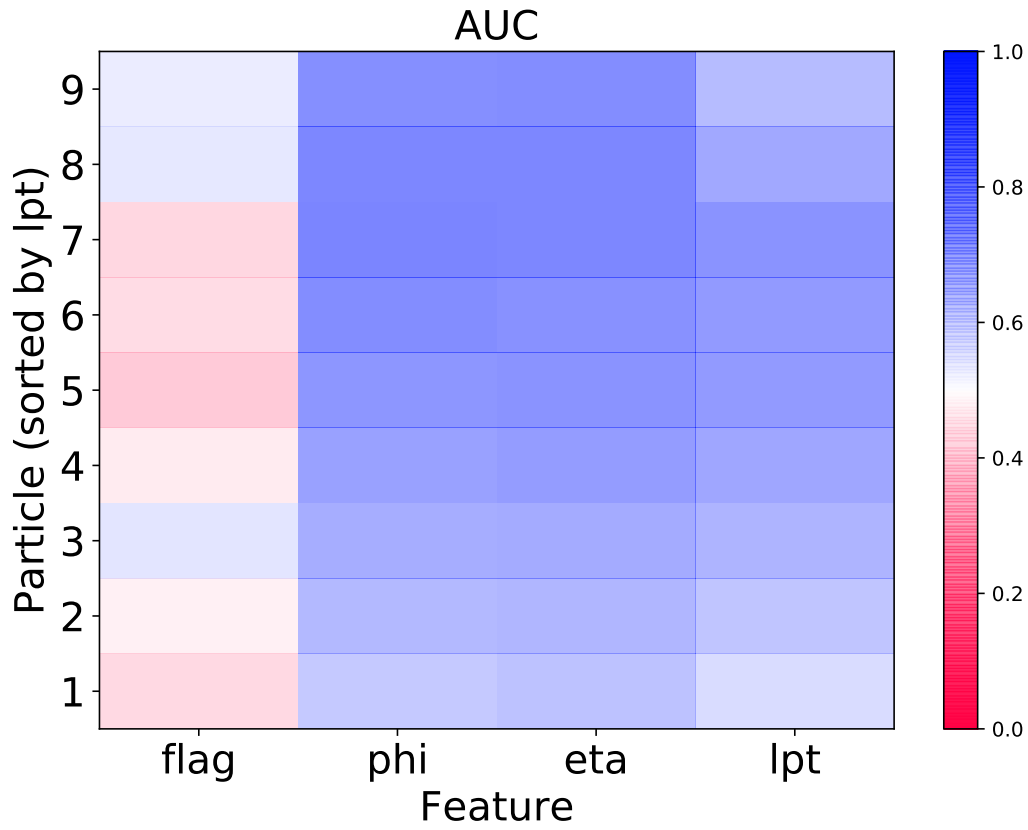


Figure 4.15: AUC feature map for 9 nodes

Also for the 9 node network, figure 4.15 shows that the most important features are the angles, they alone would reach an AUC of 0.63 here.

## 5 Apparent questions

Referenced in: [1] [7.2] [8.3] [C.2.2] [C.4.3] [D.1] [D.5.1] [F.3.3]

Given the results of the classifier introduced in the last chapter, there are two problems that limit the usefulness of these autoencoder. This chapter tries to understand them further, so that chapters 6 and 7 can solve them.

### 5.1 Scaling the network size

Referenced in: [4.8.2] [D.5.1]

The number of particles used in chapter 4 is quite low. Since we only use at most 9 particles, we usually ignore most of the jet. And when you also consider, that the 9 particle networks are generally worse than the 4 node ones in every way, this becomes a problem. Just not the way you might think right now.

#### 5.1.1 Problems in scaling

A graph update layer scales theoretically like  $(nodes \cdot params)^2$  (see chapter 7.4.2) with the number of particles *nodes* and the number of parameters *params*. This means, that by increasing the number of nodes by a factor 2, we increase the time requirements per layer by a factor<sup>49</sup> 4. Additionally, to this, you require more graph update layers for more compression steps, and so scaling becomes time-consuming. But this is not the real problem, as 30 nodes would work fine, and waiting 20 times longer than for a 9 node layer is acceptable. The real problem is the number of failing layers: the more layers you use, and the more nodes you have in them, the more probable a loss that is NAN seems to become, which would mean we need to retrain the network or accept a nonoptimal loss. Combine this with each NAN debugging step now taking days instead of minutes, and scaling is no longer easy. (You could say, that these NANs are actually a problem of the data<sup>50</sup>, since we are able to scale upwards of 50 nodes on easier data, see appendix F.1).

This does not mean that scaling on jets is impossible, but just that scaling did not seem like the best use of our time. Consider, that we actually can easily scale up the number of nodes, when we are able to give up something for it. In the following section we see 2 possibilities:

#### 5.1.2 Scaling through batches

Referenced in: [5.1.4]

Since we apparently can train 4 node networks quite well, instead of training one network with  $n$  nodes, we can train  $n/4$  networks with 4 nodes each (sorted by their transverse momentum) and just add their losses together. The price we pay here, is that particles that are too different in their momentum cannot interact at all. This means that we will probably get way less information from the relation of the particles. Also we use a l2 loss here for numerical reasons. This is shown in figure 5.1.

<sup>49</sup>Since in each layer there are either twice as many nodes or twice as many parameters.

<sup>50</sup>In the sense, that there are some events with diverging gradients.

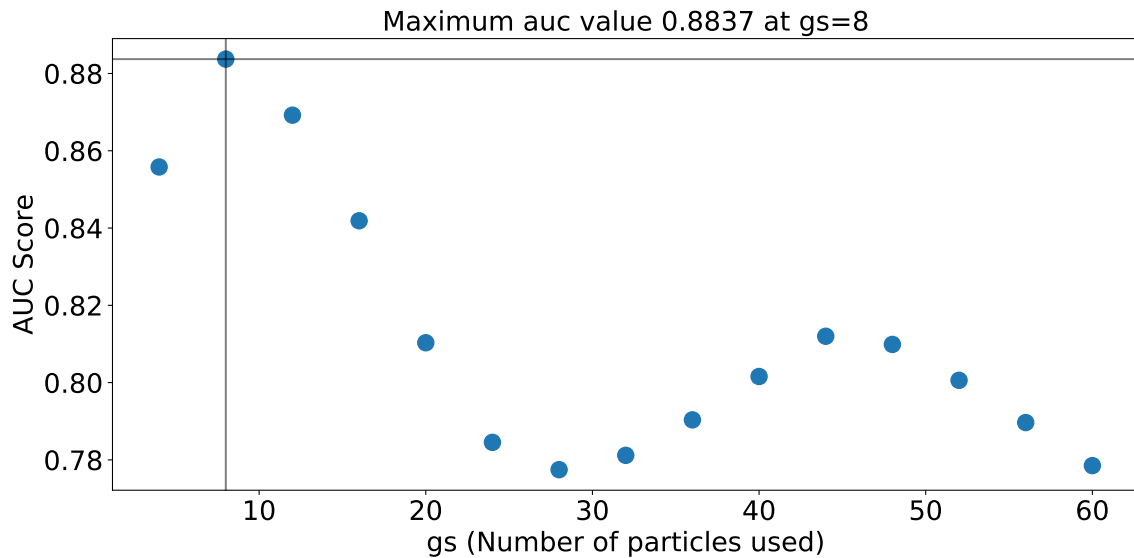


Figure 5.1: AUC score scaling through multiple batches of 4 nodes each

There you see that the AUC falls at some point by adding more batches. This means, that by adding information, we loose classification power, which is just strange.

### 5.1.3 Scaling through dense networks

Now this might be simply a feature of our batch approach. You do not only lose any interaction between certain parts of your jets, but also later batches are more and more arbitrary and thus less useful for classification. To show that this is not the central problem, here we simply solve our time dependency and our NAN problems, by using dense autoencoder instead of graph ones (see figure 5.2).

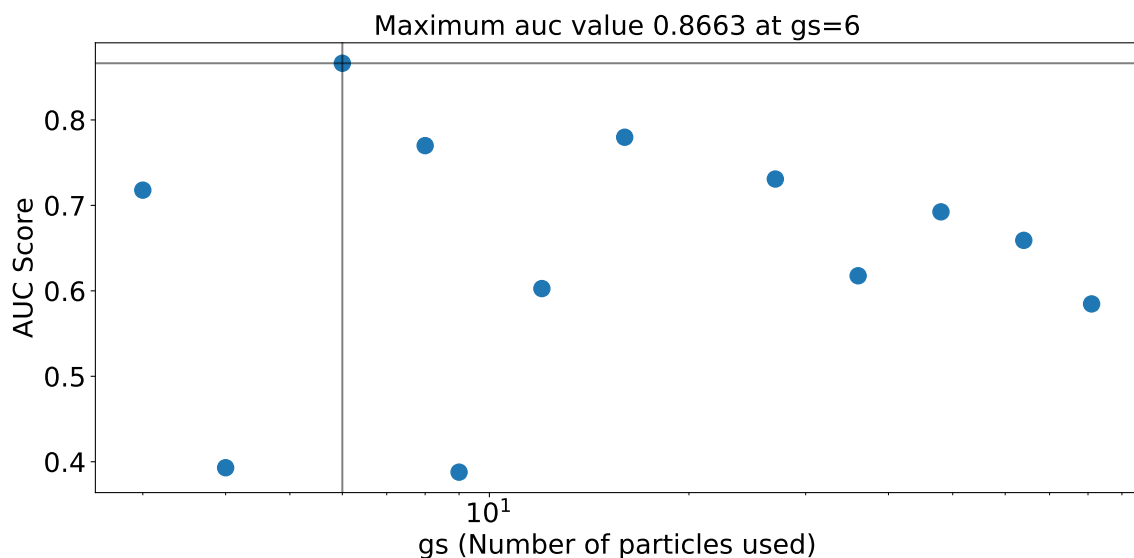


Figure 5.2: Scaling autoencoder, by using dense networks instead of of graph ones

Again, you see, that the AUC falls at some point and more information does not mean better classification. Also this relation is much less clear, but this might just be a consequence of us having less experience training dense autoencoder.

### 5.1.4 C addition

Referenced in: [3.3.2] [4.5.3] [5.1.5] [5.2.1] [9] [D.5.1] [E.3] [E.4.1] [E.6]

To understand why more information can lead to worse classifiers, consider that all of our approaches have in common, that they weight every particle in the same way, which is, as we will show in this chapter, not a very good idea.

Instinctively you can understand this as follows: Particles with lower  $p_T$ , are more random, but more random parts have a higher loss in the autoencoder, and thus matter more in the classifier. This is slightly different in image like networks. Since the loss here is the transverse momentum itself, parts of the network with higher randomness automatically have lower weight in the loss function, since their momentum is also smaller. This is another reason, why we tried to make our loss more image like in chapter 4.5.

Mathematically you can model this by considering features of the following kind: Given two gaussian distributions like in figure 5.3, with variable overlapping, the gaussian peaks can describe background and signal respectively. The quality of the described feature can be seen as the inverse of the overlapping fraction. This is basically what an AUC score calculates<sup>51</sup>, and so we can optimize the combination of two features by combining two different double gaussian peaks in a way that minimizes their overlapping fraction.

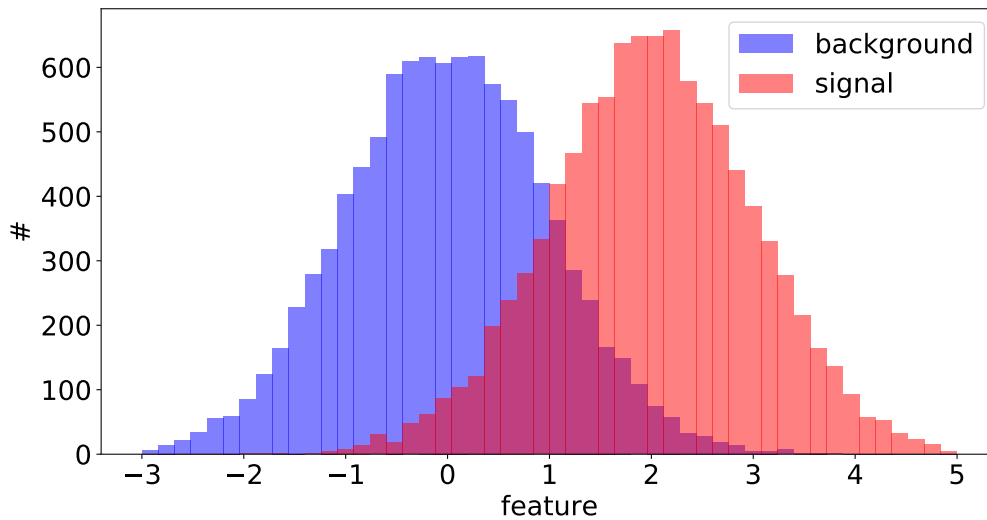


Figure 5.3: An example of how we model AUC as a function of the overlapp of two gaussian peaks

To do this, first notice, that the quality of one of those double peaks is translation invariant, as well as scale invariant<sup>52</sup>. This means, we can set two values to be fixed: We choose here the mean values of those peaks to be  $\mu_0 = 0$  and  $\mu_1 = 1$  and for simplicity we also set the width of both peaks to be the same ( $\sigma_0 = \sigma_1$ )<sup>53</sup>.

Now we want to add two double peaks with some constant relative factor<sup>54</sup>:

$$f = c \cdot f_1 + f_0 \quad (5.1)$$

<sup>51</sup>The AUC score is a monotonously falling function of the overlapping fraction.

<sup>52</sup>To be more precise, invariant under any monotonous transformation  $f(x)$ .

<sup>53</sup>This assumption does really effect the final result, but simplifies the calculation. You get the exact result, when you set the new sigma to the quadratic sum of the original widths:  $\sigma^2 = \sigma_0^2 + \sigma_1^2$ .

<sup>54</sup>You might ask if this is the most generall approach of combining two features, and in general it is not, but the invariance of those features under fairly general transformation make most other combinations useless, and something like a factor that depends on the current position would break our assumption of gaussian peaks, and thus complicate the calculation for a probably quite low difference.

The result is again a double gaussian peak feature, and thus we can evaluate it's quality by the same measure: The new means are given by  $\mu_0 = 0$  and  $\mu_1 = c + 1$ , while the new width are given by  $\sigma^2 = c^2 \cdot \sigma_2^2 + \sigma_1^2$ . To be able to compare the overlapping fraction, we use scale invariance to assert  $\mu_1 = 1$ , and thus the width, can be minimized to find the optimal combination of two features. This width is given by:

$$\frac{\sqrt{c^2 \cdot \sigma_2^2 + \sigma_1^2}}{c + 1} \quad (5.2)$$

which has the same minima as (with  $q = \sigma_2/\sigma_1$ )

$$\frac{(c \cdot q)^2 + 1}{(c + 1)^2} \quad (5.3)$$

and which is minimal at

$$c = \frac{1}{q^2} \quad (5.4)$$

So as expected, the bigger the width of one feature is (the more random it is), the less it should contribute. Note that this exact relation is only true, when the mean of the signal is constantly 1, and that if it is not true, one has to add another factor  $\mu_1/\mu_2$  to the relation.

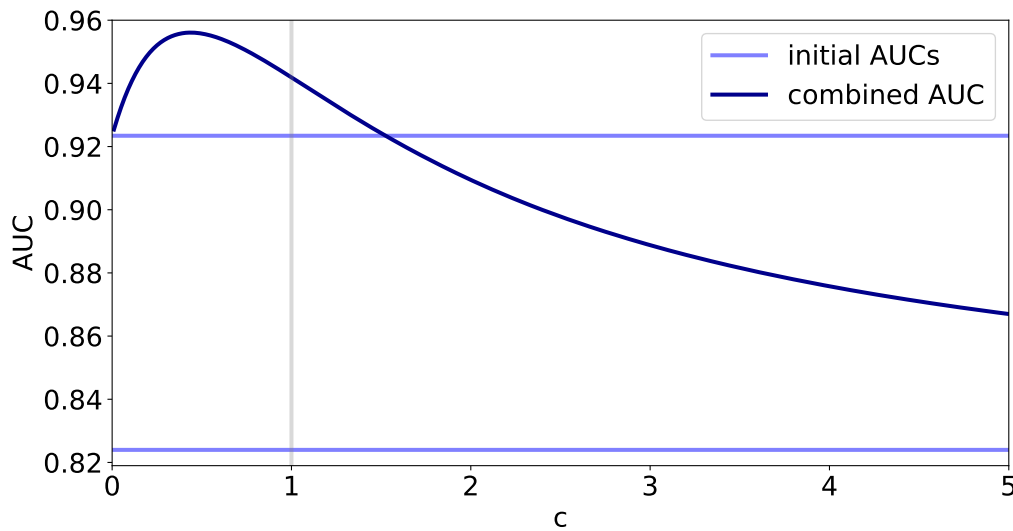


Figure 5.4: AUC as function of  $c$  for two random gaussian double peaks that alone would reach AUCs represented by the horizontal lines

To test this relation, consider figure 5.4. It shows that there is an optimal  $c$  value, at which you can combine two features as good as possible. Also as long as you are close to this optimal value, the resulting quality is still better than either single classifier. This is no longer the case when you go too far away from the optimum. Then adding more information can actually hurt the original quality. This might explain our problem of more information resulting in less good classification. To test this on jets, we use the batchlike autoencoder from chapter 5.1.2. Instead of simply adding these batches together, we use the derived formula for adding distributions (This is actually done in an unsupervised way. The problem here would normally be the mean values of the signal distribution, but we can approximate them, as well as the width of the

distributions by the network loss, and thus multiply each feature by the inverse of the third power of this loss ( $\frac{1}{\text{loss}^3}$ )<sup>55</sup>) This can improve our batch approach by about 1%.

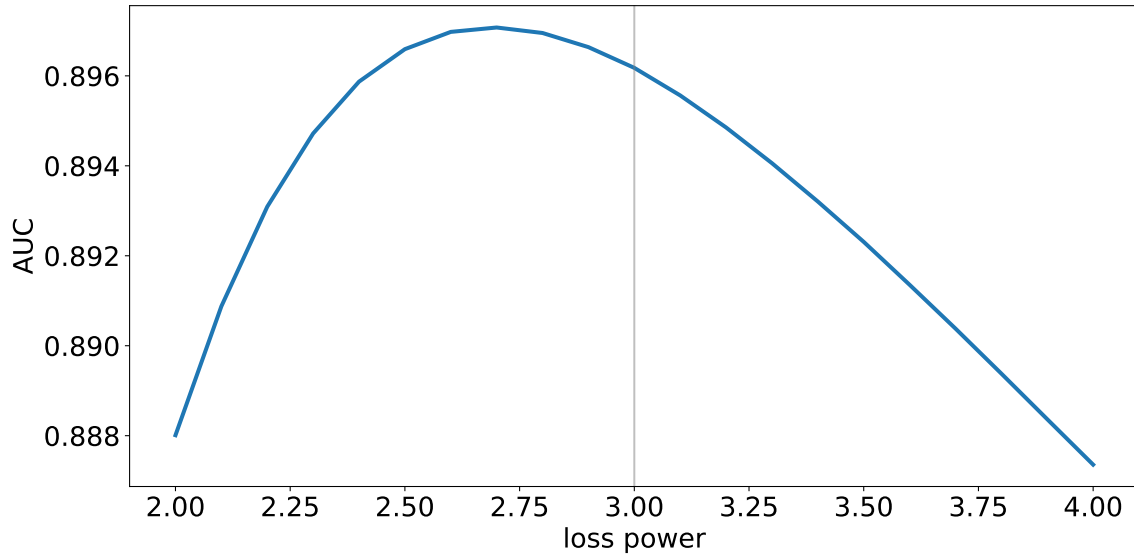


Figure 5.5: AUC score as a function of the loss power ( $-3$ ) for parts of a QCD jet, using 5 4 node networks combined in a way defined by the loss power

You might see some slight deviations from the expected result in figure 5.5, but generally using the derived formula, a power of 3 seems to result in a good AUC value, proving our method. If you assume that the difference is not just statistical, the difference might be explained by carefully considering our quite extensive assumptions. We do this in appendix E.6.

### 5.1.5 Scaling through losses

Referenced in: [7.4.2]

Using only a few particles, addition does not seem to be such a problem, but this does not help the training of bigger networks.

One way to achieve this, would be to redefine the loss function. You could do this in such a way, that the loss of any particle gets multiplied by the factors used in the split networks. The problem here is that this changes the focus of the autoencoder quite drastically: Since then it can make more errors in the later particles, it will do so, making the later particles less useful for classification. And also making the first particles more useless, since the focus is now lying on them, making their reconstruction better, up to a point at which their reconstruction is too good to find any meaningful difference between background and signal. Another idea might be to just apply this loss weighing in the evaluation phase, and not in training. This definitely helps, but in our tries does not seem to be enough, since the same effect as before works now in the opposite way: Particles with lower  $p_T$  have a higher inaccuracy, that translates to a higher loss for them, and the autoencoder focussing more on them, making the lower particles and the higher particles less useful. So it seems that this is an optimization problem: There still seems to be an optimal loss weighing that gives optimal contribution to each part, but finding this combination is not trivial. We tried multiple functions including a loss that weights using the index of the particle or a loss that is weighted directly by the transverse momentum, but we

<sup>55</sup>To be more precise, its square root (since the loss is quadrated): this works, since the l2 loss of the one sided training is the variance of the first peak, and by noticing that both widths are not too differently in practice, and that the higher the loss is, the second peak seems to move away too. We test this assumption more in appendix E.4.

have not found anything that generally works, and weighted networks always seem to result in worse looking reconstruction.

You could even argue, that finding a good weighting is not actually worth it, since you need to look at a lot of networks and compare their classification score. And by doing this a lot, this means that the anomalies you use loose generality, and you find a network that is only able to find this special kind of anomaly: The information in your anomaly dataset leaks into your model setup, and more training results in less general models. That does not mean that a good scaled network is impossible, the ideas from chapters 5.1.4, 4.5 or 7, might be a good approach, but this is not an easy task, even though chapter 7.4.2 solves the numerical problems.

## 5.2 Simplicity and invertibility

### 5.2.1 Simplicity

Referenced in: [4.6.4] [5.2.2] [6.1.2] [6.2.3] [7.1.1] [8.1] [9] [9.1]

One thing you can do, by comparing values directly, is to look only at parts of the loss. This allows you to define qualities for each part of the input space. As a reminder of chapter 3.3.2, we show them here as AUC maps: each AUC value is one colored box, which is deeply blue for an AUC of 1, completely red for an AUC of 0 and white for an AUC of  $1/2$ .

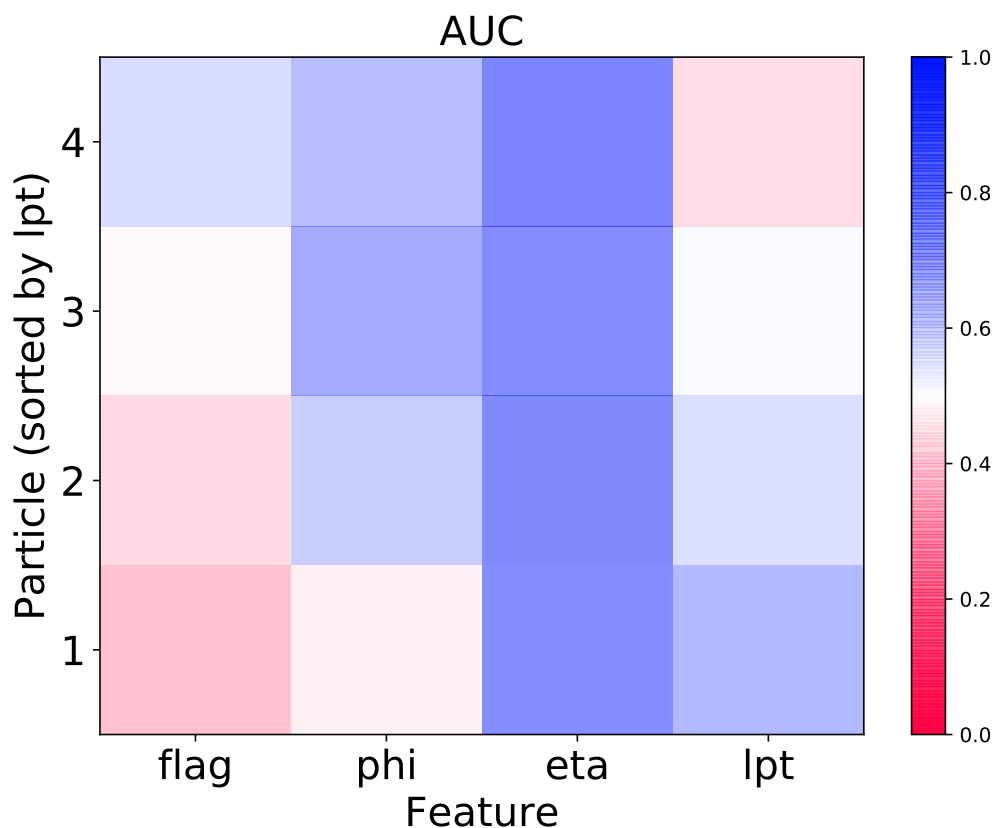


Figure 5.6: AUC map for a simple network

As you see in figure 5.6, the classification quality is mostly set in the angular part. This is fairly common, as there are networks, in which the nonangular parts are partially red.

On the other hand, if you look at the loss distribution in figure 5.7, the angles are the variables known the least.

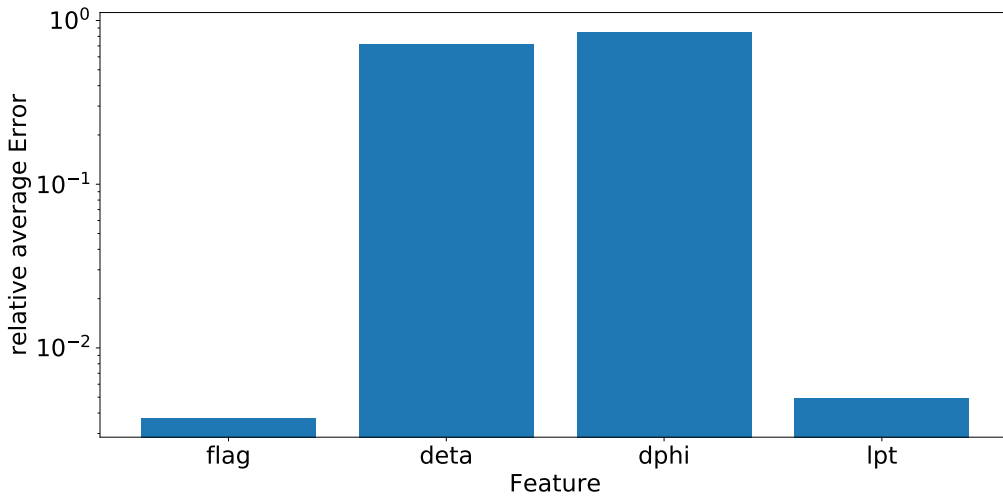


Figure 5.7: Average relative error by feature ( $\frac{\text{mean}(|x-f(x)|)}{\text{mean}(|x|)}$ )

This is a bit strange: The thing the network seems to not care about, is the thing that the classifier considers most useful.

We understand this as follows: If you look at the 2d histogram of the angular distribution, there is a clear difference between top and QCD events.

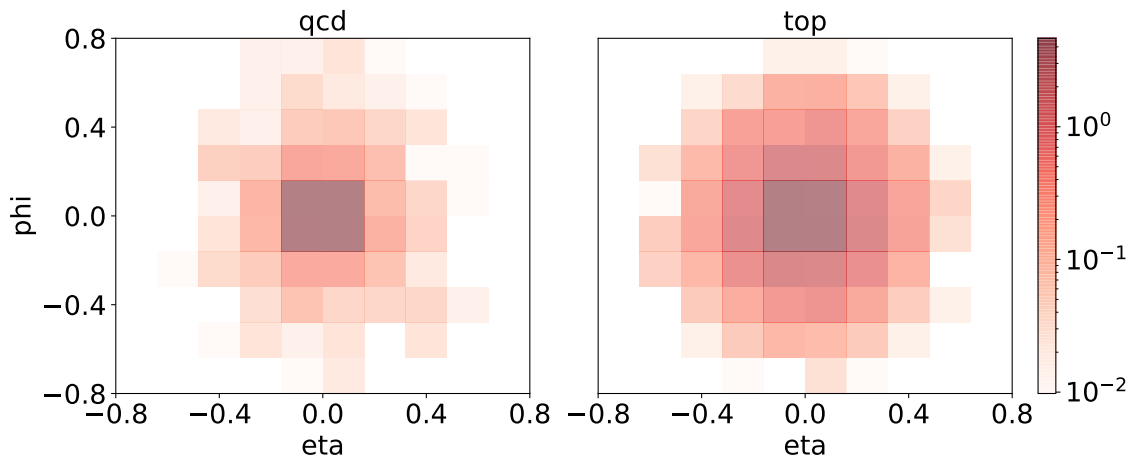


Figure 5.8: 2d histogram of angles comparing QCD vs top, here for example for the particle with the fourth highest transverse momentum

You can see in figure 5.8, the width of top jets is much higher than the width of QCD jets, so by comparing both angles to zero, the top jets statistically have a higher loss than the QCD jets, and this can be used to differentiate between them easily. This is used especially since our neuronal networks tend to reproduce mean values. So how useful is this difference, and how much better does the network do than this relatively trivial separator? First, a model that only uses its angles to classify jets, works very similar to a model that also adds the loss in  $lp_T$  (and flag), so we can assume that this is truly just a problem of angles: Now given a model that just outputs 0 for the angles and only considers the angles in the loss, you scale it like in figure 5.9.

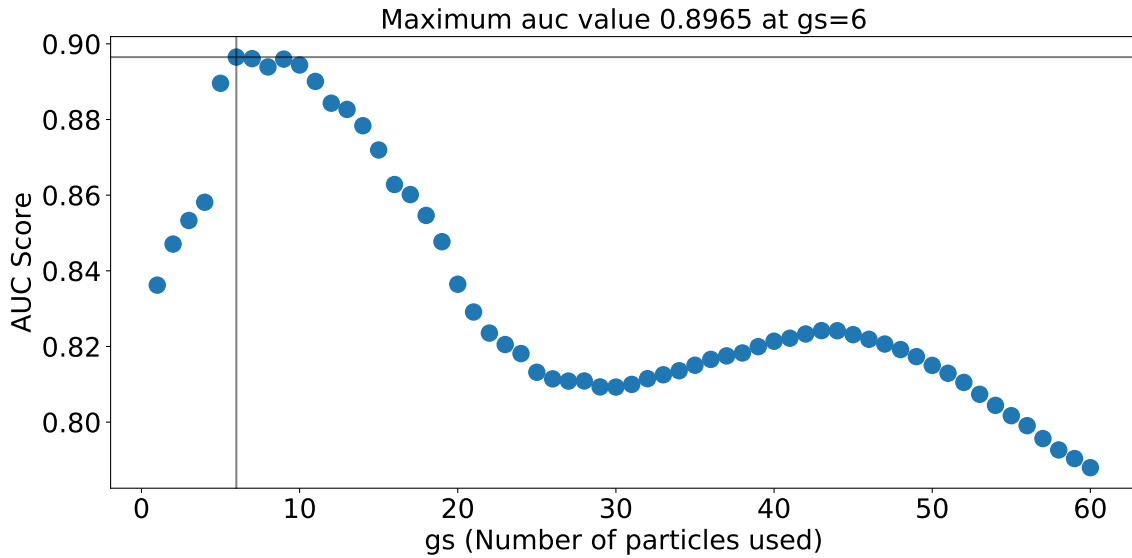


Figure 5.9: Trivial width comparing angular scaling. Here we do not split each network in batches of 4 anymore, but simply calculate the AUC for each number of nodes up to 60.

As you see, for a low number of particles, this works fairly well. But at some point, more information does not mean better classification, and the quality drops. But this is a problem we already know about (see chapter 5.1.4) and can simply solve through  $c$  addition. So when you add each particle together weighted accordingly to their loss (its angular difference to 0), you gain a better scaling behavior (compare figures 5.10 and 5.9).

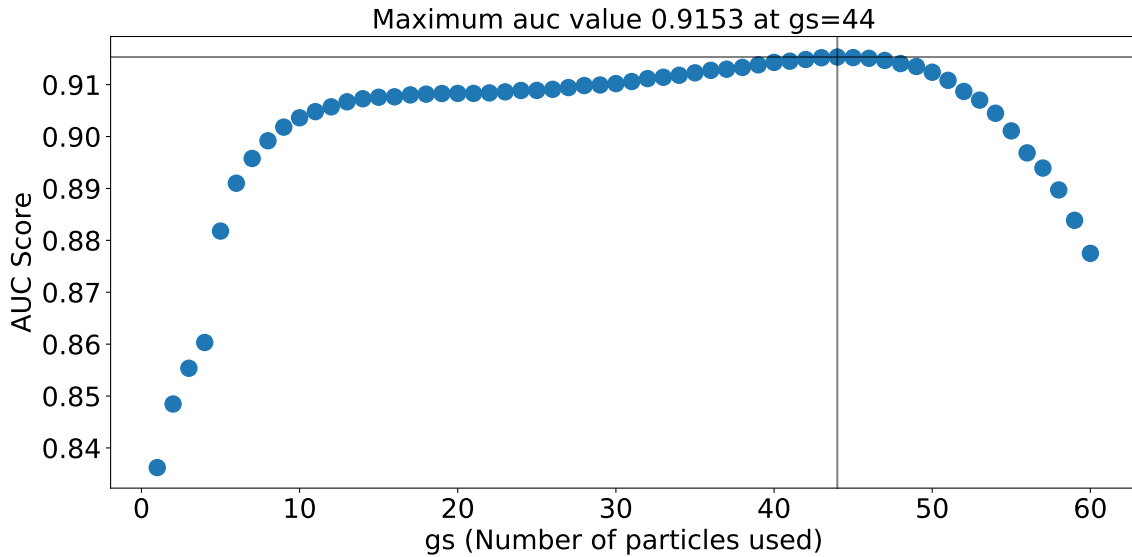


Figure 5.10: Trivial width comparing angular scaling with  $c$  addition. The reason for the falloff at the end might be the different shape in later indices of missing particles or the assumptions tested in appendix E.4

This better classifier reaches an AUC of over 0.9155, which is comparable to the best anomaly detection networks, for example QCDorWhat[24] reaches 0.93 but on slightly different data, while the work of Thorben Finke[20] reached 0.908 on the same data. You could ask yourself what the value of those complicated models is, if they only improve the AUC by at most single percentage differences.

More importantly you also cannot assume that new physics has the same angular distribution difference, as QCD compared to top, making this alternative model useless in the task of finding new physics<sup>56</sup>, so the question of interest is just: do complicated models contain something more than this trivial difference? And unfortunately this is very hard to test. C addition allows you to estimate the effect any small additional AUC would have, and an uncorrelated AUC of about 0.6, optimally combined would only improve an AUC score of 0.9 to 0.904, while 0.7 would improve it up to 0.917. So both improvements would probably be nearly immeasurable. This means that there might be some hidden effect in a model, that allows them to find new physics<sup>57</sup>. What we can say, is that the networks we looked at so far, probably don't do anything more than looking at angular information<sup>58</sup> and thus their fairly good AUC score is just a consequence of our trivial difference, and thus most likely completely useless for finding new physics.

What we want to do in the following is to force our network to learn something nontrivial, and thus actually to create a correlation between how the network works on top jet anomalies and how it would behave on new physics.

Probably the most important result of this trivial model, is the effect it has on how to evaluate a model. We already talked about why just choosing a model that has a good AUC is a bad idea in chapter 4.6, but here this could probably not be clearer: Since we tried to train a model to be a great classifier, when we changed the initial normalization, we choose those, that makes the network ignore angles and focus on  $lp_T$ . This seemed useful, since this makes the model more like the trivial one and thus get a good AUC. But this also means, that we don't have a good autoencoder anymore, since a worse reconstruction can actually improve the quality of the network. You can also see this in the number of nodes we use: 4 node networks seemed to result in good classifiers, which matches the pattern in the first 4 nodes, which you can see in figures 5.10 and 5.9.

---

<sup>56</sup>Or at least useless unless you search for one specific kind of abnormal data, there are some examples showing other kinds of abnormal data behaving completely differently in 8.

<sup>57</sup>We assume here a lot: first, that in  $lp_T$  there is potential to differentiate all kinds of new physics, that this potential is used perfectly by an algorithm that did not do this for angles, and also, maybe most improbable, that the loss is combined perfectly, and there is no confusion from the angular part at all.

<sup>58</sup>Since the classifier is dominated by the angular part, and it does not deteriorate when you remove momenta, and the resulting classifier again improves when you replace the returned angles by zero.

### 5.2.2 Invertibility

Referenced in: [8.1]

Given a classifier that if trained on QCD, finds top jets anomalies, you should always ask yourself, if this is just a feature of your data, as shown in chapter 5.2.1 or if this is something more general. One easy way to test this, is just to switch the meaning of signal and background: Can an autoencoder trained on top jets classify QCD jets as anomalies. (To keep the usual plots being easily readable, we keep QCD as background, which results in an AUC score of 0 being optimal for those switched networks). This does not yield the desired results at all. You see in figure 5.11 that a model trained on top jets, is still a valid classifier for QCD jets.

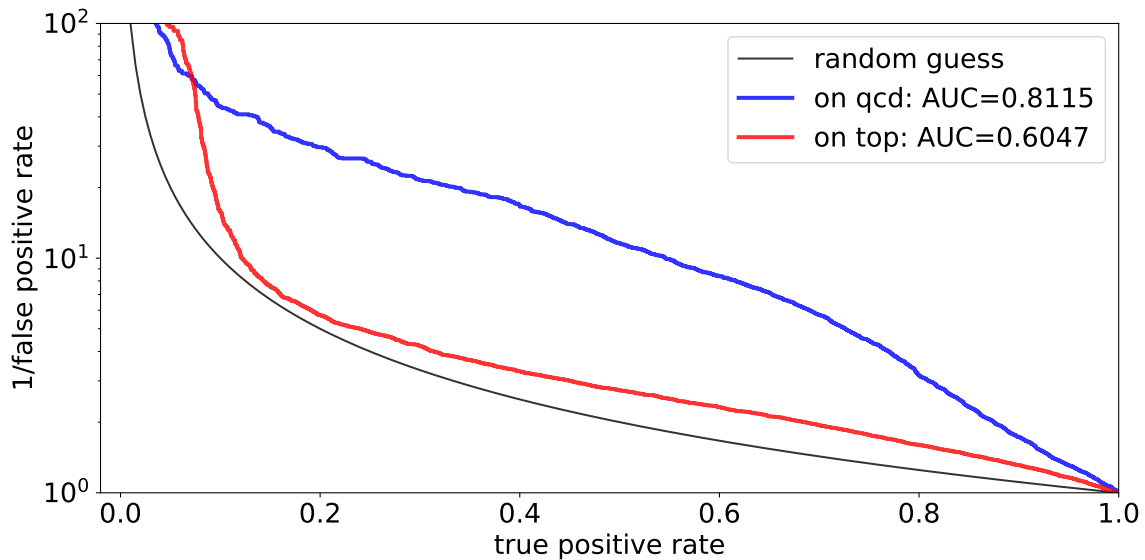


Figure 5.11: Roc curves for the invertibility of a 4 node model

This could actually been expected: As shown in the last chapter 5.2.1, our networks that have a good AUC score, focus mostly on the difference between the angles and 0, and since this model does not depend on the attributes of the training data, changing the training data does not alter them, and so it does not affect their classification. In fact, by choosing a model that focuses even more on the angular size, you can create models that are completely independent of the training data.

All of this is obviously quite problematic, which is why the next two chapters (6 and 7) suggest solutions, and after doing so, chapter 8 focusses entirely on other datasets and their invertibility to make sure that our solution works in general.

## 6 Normalization

Referenced in: [1] [5] [5.2.2] [9] [D.1]

### 6.1 Introudicing normalization for autoencoder

Referenced in: [4.6.4] [6.2.3] [7.4.2] [8.3] [D.5.1]

When we remove trivial features from our data, we can prevent our networks from only learning those. This is what we try in this chapter.

#### 6.1.1 The meaning of complexity

Since models seem to be not invertible since they contain a good trivial model (see chapter 5.2.1), it stands to reason that we might get an invertible model by removing this trivial model from it. This would make these models more general, since they cannot rely on trivial information to perform well. We can remove the kind of feature, that seems to allow for this trivial comparison, by normating our input, since chapter 5.2.1 shows, that the width of the angular input distribution is the trivial feature<sup>59</sup>. This method has one obvious drawback: Not only do we actively remove information, which might hurt the performance, but we remove the information, that is most useful for the classification. This means, even if the resulting classifier is invertible, it will look way worse than the trivial one before. This does not mean that these networks are less useful, as we trade quality on the task of QCD v top jets against the generality that is desperately needed in the networks of chapter 4, but an approach not loosing quality would clearly be better (see out second solution in chapter 7).

#### 6.1.2 How to normalise an autoencoder

One thing, we did not realize before trying to normalize the input data points, is that simply demanding that the mean is zero and the standard deviation is one, just does not work. This might be an effect that is most important when we talk about small networks<sup>60</sup>, but is still somewhat of an effect in every network, and becomes especcially important in chapters 7.1 and 7. The problem is, that by demanding a value to be fixed, we remove the features from the input space, and by having an autoencoder that only reduces 12+flag information onto 9 values, this means, we allow the network to trivially learn 3 informations per set feature<sup>61</sup>, and so by setting the standart deviation and mean to be fixed, the autoencoder can trivially learn to compress 12+flag onto 6 values<sup>62</sup>, which is below the size of compression space. In practice this is not so easy as there is no guarantee that these minima is found, as the graph structure does not necessarily help this kind of transformation(see appendix D.2), but training this kind of network definitely does not result in the model gaining any classification power. This can be seen in the corresponding feature map (figure 6.1)

<sup>59</sup>It would be enough to normate angular values, but to gain generality, we also normate the momentum information.

<sup>60</sup>Networks with a low amount of input particles.

<sup>61</sup>3, since there are 3 variables which mean and or standart deviation we fix, normalizing flag does not seem to be a good idea.

<sup>62</sup>Ignoring flag for now, the remaining three values are always enough to encode 4 flag values, since the first flag values are neirly always one (the jet with the lowest number of particles has 3 particles in our trainingsset).

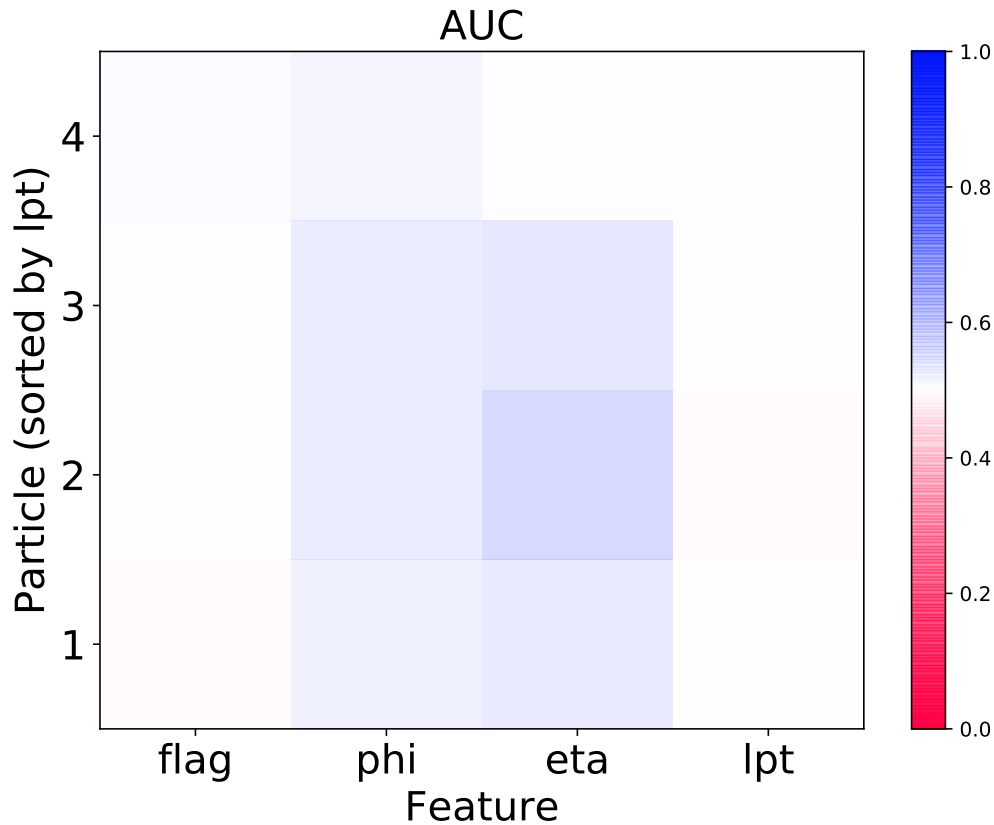


Figure 6.1: Aucmap for normally normalized networks, showing not much useful being learned. We train here on top jets to test the invertibility

This seems as if there is a trivial solution: just reduce the compression size accordingly, but this has three problems

- First, it is not completely trivial to misuse the normalization (Think of the standard deviation, there is a formula giving you information about the 4th value, given the first three. But even if we ignore the mean as being 0, this formula still involves squares and roots, which the network has to learn, and even then, there are always two possibilities for the resulting value.). So assuming that this is trivial, and that the network will always learn it guaranteed, would be wrong.
- Even if this is learned, this would not be enough: the network still has to compress this information further and this can lead to situations in which the network has to decide between learning the easy compression and the learning the interesting compression. In these situations it will probably always learn the trivial one.
- Trying to compress data with removed information further is not as easy as compressing non-removed information. Think of two values distributed between 0 and 10 For example 4 and 6, or, after setting the mean to be 0:  $-1$  and  $1$ . In 4 and 6 the network can still decide to just average both values and get a mediocre prediction of 5, which still describes these values in a way. But if after removing the mean, the network still averages both values, the predicted 0 is fairly unproductive<sup>63</sup>. We conclude, that simply subtracting

<sup>63</sup>Note, that the difference between a good normalization and a bad one is just physical intuition. For example, we still set the mean of the jet angles to be zero, just because the direction relative to the measurement should be unimportant.

each fixed value from the compression size does not work, as we would expect a less good classifier.

So what we need, is a better way of normalizing the input data. From our thoughts above, we suggest that this new method should satisfy these three conditions:

- Translation invariance:  $n(x) = n(a + x)$ .
- Scale invariance:  $n(x) = n(a \cdot x)$ .
- No fixed features: you can not write any  $f(x)$  so that  $f(n(x)) = 0$  for every  $x$  is given.

The first two rules are obvious, since we want to use this, to remove any size information, and the third rule would solve the problem of an autoencoder focussing on normalization artefacts<sup>64</sup>.

All three rules<sup>65</sup> are solved by the following 3 normalization steps ( $x$  is the input,  $n$  the output of the normalization method)

$$y = x - \text{mean}(x) \quad (6.1)$$

$$z = y - \text{mean}(|y|) \quad (6.2)$$

$$n = \frac{z}{\max(|y|) + 0.001} \quad (6.3)$$

Here the definitions of  $y$  and  $n$  assert translation and scale invariance respectively, while  $n$  and  $z$  remove any kind of artifacts. Why they do this can be easily understood for  $n$ , by diving through the maximum value instead of the standard deviation: The only relation given is, that if none of the first three values is either 1 or  $-1$ , the last value is either 1 or  $-1$ . And even if we ignore that misusing this relation would be quite complicated to implement for a neural network<sup>66</sup> there is no way to differentiate between 1 and  $-1$  in general<sup>67</sup>. Also dividing through the maximum value is generally a good idea compared to dividing through the standart deviation, since it only divides by zero when every value is zero, and not when every value is the same, which is more probable<sup>68</sup>. Also dividing by the maximum is generally a bit faster, while resulting in fewer NaNs (appendix B.2.2). The other definition is less easily understandable: First note, that it does not violate the first two rules, since  $y$  is already translation invariant,  $z$  is too, and since every  $z(y)$  is scale invariant, Scale invariance is also given<sup>69</sup>. Now consider only the definition of  $z$  and two different vectors  $y$  given by  $[0 \ 0]$  and  $[1 \ -1]$ . Both result in the same first component, but different second components, so there is no clear function giving the second value as a function of the first one, and thus rule 3 is satisfied.

<sup>64</sup>This last rule would actually already be solved by demanding the standart deviation to be constant.

<sup>65</sup>Except for scale invariance with  $a \leq 0$ .

<sup>66</sup>Neuronal network work with uncertainties: if you have a value of 0.997, is there still a 1 remaining?

<sup>67</sup>You could say, that this normalization uses the problem of mean reproducing networks (appendix B.6) to its benefit, by making the errors of a 1 or  $-1$  guessing network bigger than every other possible distance. But this is a bit more complicated through the definition of  $z$ , since this kind of outputs have a clearly negative bias.

<sup>68</sup>There is a small constant in our definition to remove this divergences, but this still removes some big gradients.

<sup>69</sup>You might notice, that this is only the case for positive multipliers. You could also argue, that we only want to remove nonphysical information, and since for  $lp_T$  this is not a problem (since  $lp_T$  is positive and it has a peculiar shape), this is only interesting for angular information, and parity is broken.

## 6.2 Using this normalization

Using this kind of normalization, 4 node networks are invertible. And not only this, but also most features are invertible (compare the feature maps in figures 6.2 and 6.3).

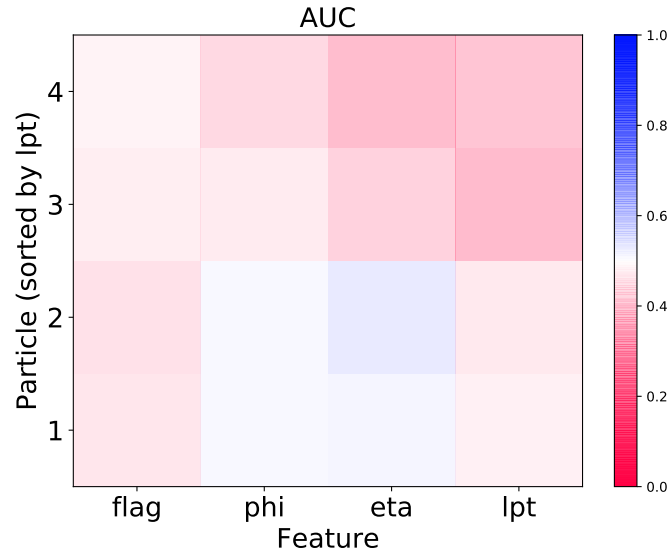


Figure 6.2: Invertible 4node network auc maps achieved by a normalization. Here trained on top jets

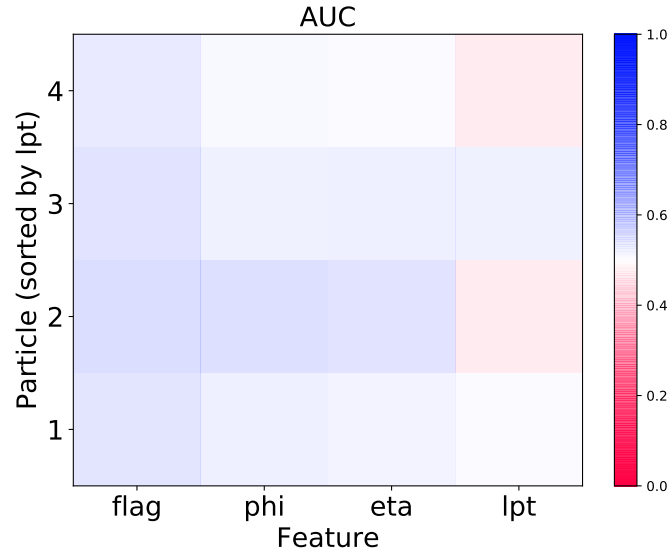


Figure 6.3: Invertible 4node network auc maps achieved by a normalization. Here trained on qcd jets

But figure 6.4 shows that the quality suffers (and that both networks only differ for true positive rates above 0.2).

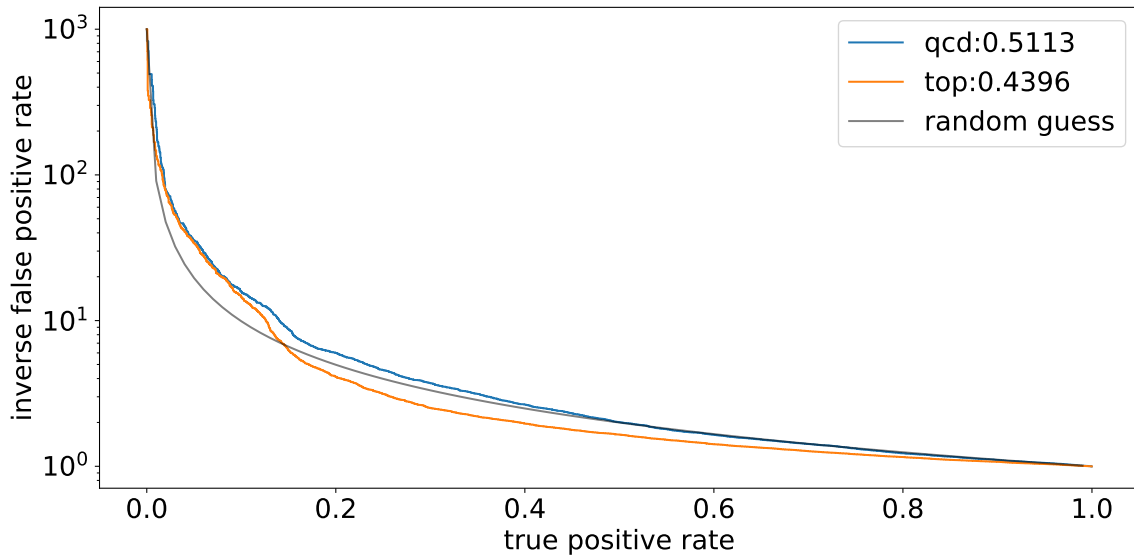


Figure 6.4: Double roc curve for the invertibility of a normalized networks

and there are other consequences of the fact that this network actually has to learn something nontrivial: First, we were forced to increase the size of the compressed feature space from 5 to 9. This makes sense, as a network that compares angles to zero, has to just reconstruct zeros in each angle, and thus only has to save  $lp_T$ <sup>70</sup>, needing only a smaller latent space. Also networks, that before were very reproducible in their training<sup>71</sup> are now less stable, and often vary their loss over about one order of magnitude. Interestingly, this variation shows a clear relation between the loss, and the classification quality. Figure 6.5 shows that this relation even extends to different training setups.

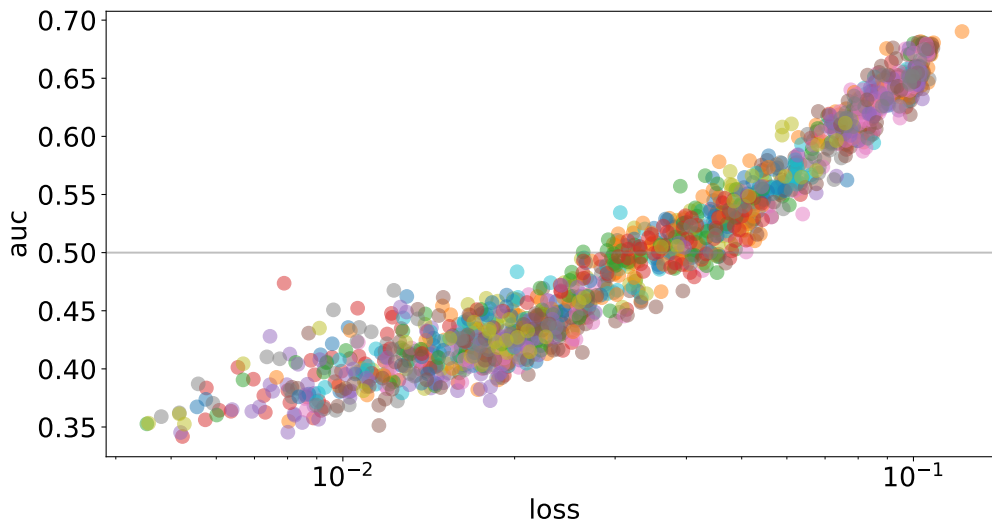


Figure 6.5: More than 1500 Models, showing a clear relation between the network loss and the AUC score, each color represents slightly different training setups. It would be linear if the x-axis would be linear

This relation is very useful, since it means, that finding a better autoencoder, automatically

<sup>70</sup>And maybe flag, but as seen later in this chapter and more in chapter 7.1, this is actually usually not the case.

<sup>71</sup>Which makes sense, as they always just needed to learn to ignored the angles.

results in a better classifier, and we thus can focus completely on improving the autoencoder. Also by looking at this relation, we are able to justify the new compression size in figure 6.6, since this is the first compression size, at which the network becomes invertible.

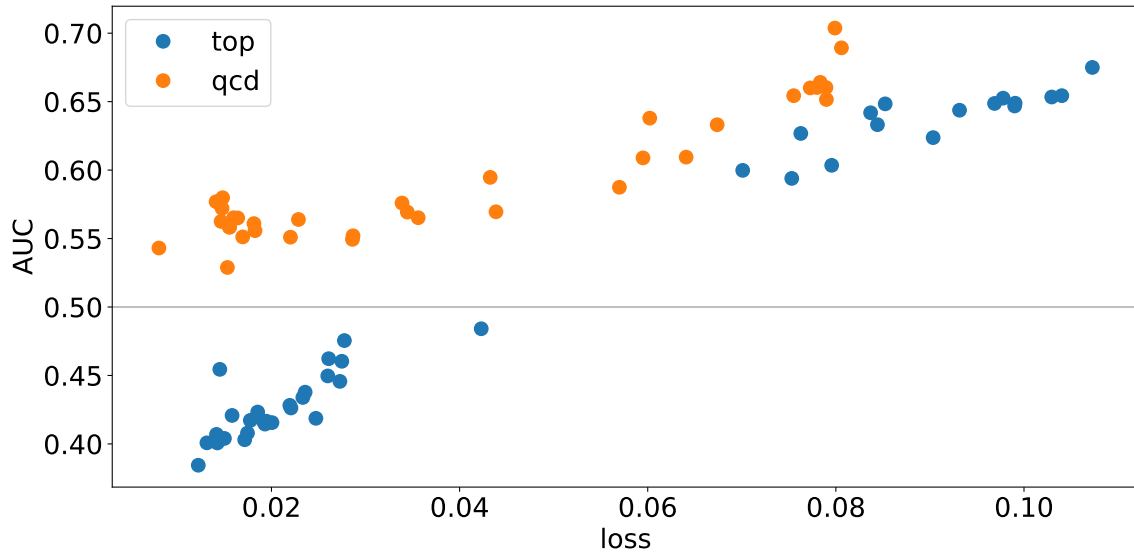


Figure 6.6: Invertibility for compressing 4 node networks into a 9 dimensional latent space. Notice the jump of the top AUC from above 0.5 to below 0.5 for lower losses. It means that when can use our loss to see if a trained network generates an invertible classifier. For compression sizes lower than 9 this jump does not appear

This variation of the networks is the reason we started training as long as described in chapter 4.4, as training for longer is required to get reproducibel networks:

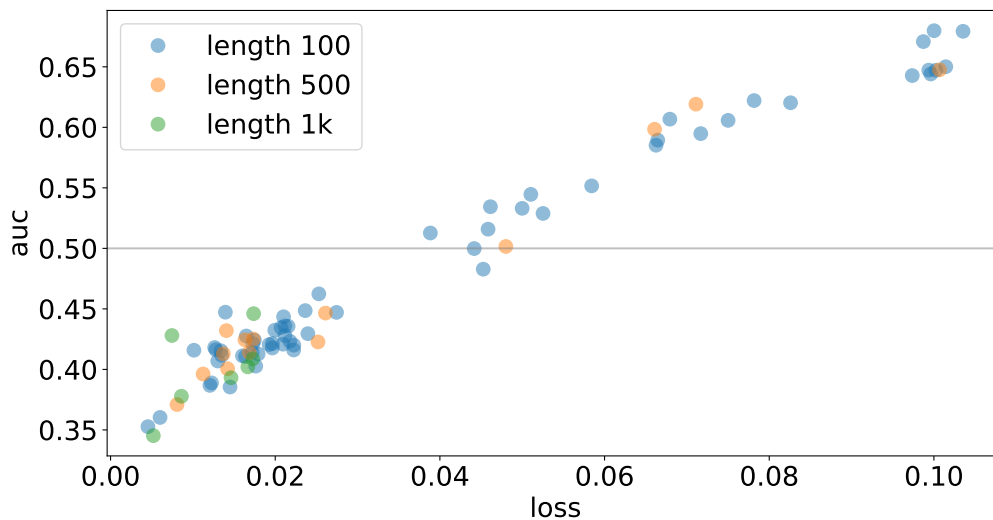


Figure 6.7: Reproducability comparison of for different training lengths

As you see in figure 6.7, training for more epochs makes the network more predictable. The most predictable results we get by not only training for at least 1000 epochs, but afterwards also until the loss does not increase for 5000 epochs. This takes a bit too much time, so in the following we use 500 epochs and we wait until 100 further epochs don't improve the loss anymore.

### 6.2.1 Improving the AUC scores for normalized networks

Referenced in: [7.4.2] [D.5.1]

These initial normalized networks are not very good. This might be what we expected, since we remove trivial information, but we still are able to improve on them quite a bit. Namely by using the exact model setups and training parameters from 4.4 with one additional normalization layer before the first comparison value<sup>72</sup>. Using this we are able to improve the network trained on top up to 0.377.

### 6.2.2 Scaling in normalized networks

Sadly this normalization does not change scaling problems too much. Bigger networks still contain more trivial information, as the number of parameters fixed is constant (see figure 7.4.2). And even when using batches to scale, the invertibility is just a feature of the first batch, as figure 6.8 suggests.

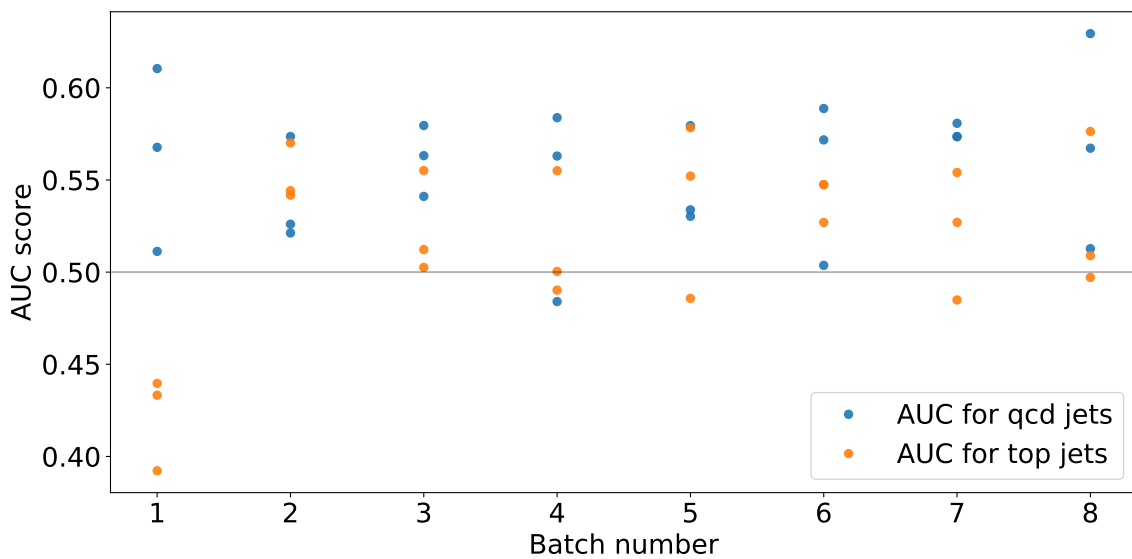


Figure 6.8: AUC values for higher normalized batches by their training data

### 6.2.3 Improving the normalization even further

After seeing what an effect some kind of normalization can have, we are not completely satisfied anymore with the normalized feature maps like in figure 6.9.

<sup>72</sup>You might be quite a bit confused, why we chose other models but those that work well for unnormalized networks to test our normalization, but this is just a problem of way to many just slightly varying network setups: We used more quite different unnormalized networks, but since learning zeros does not depend to much on network parameters, we simply use the new normalized network setups for both networks, to not need to explain both.

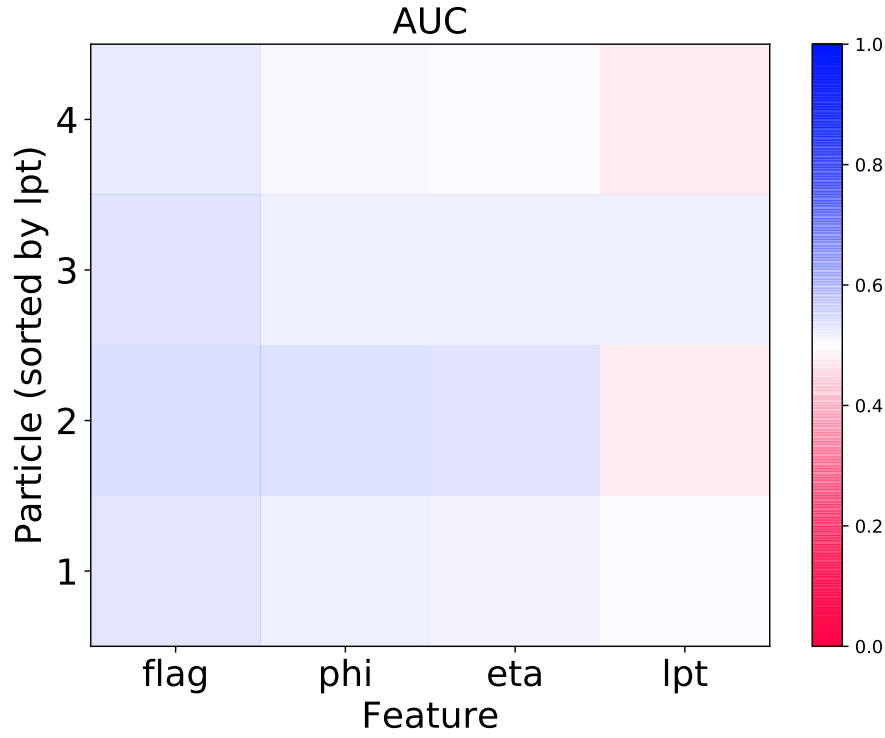


Figure 6.9: AUC feature map for normalized top trained networks

consider the highest  $p_T$  Value (the lower right corner). While being the generally most interesting particle, there is no classification power in it at all, and by looking at its distribution (figure 6.10) it becomes clear why

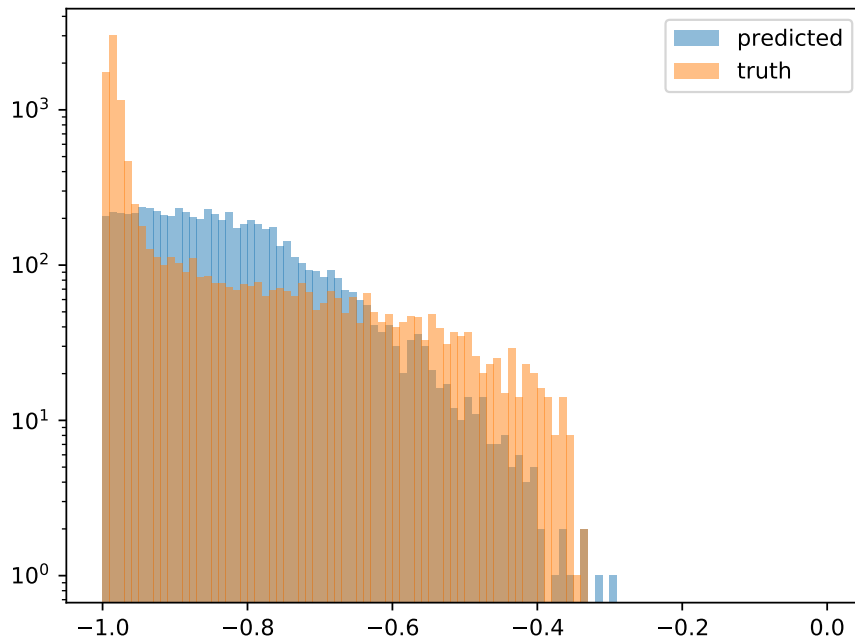


Figure 6.10: Distribution of the transverse momentum of the first particle

Its values are basically constant, so its input is the same as the flag values (first column),

from which we don't expect any physically useful information.

To solve this, consider the following: Since  $lp_T$  mostly has the same structure<sup>73</sup>, most jets transverse momentum get divided by the first one, resulting in it always having the same value. We can fix this by replacing the definition of  $n$  in chapter 6.1 to be:

$$n = \frac{2 \cdot z}{\max(|z|) + \text{mean}(|z|)} \quad (6.4)$$

removing the need to set one value to either positive or negative one, and thus making the highest value in  $lp_T$  actually useful, and as you see in figure 6.11, this removes the difference in the  $lp_T$  AUC scores.

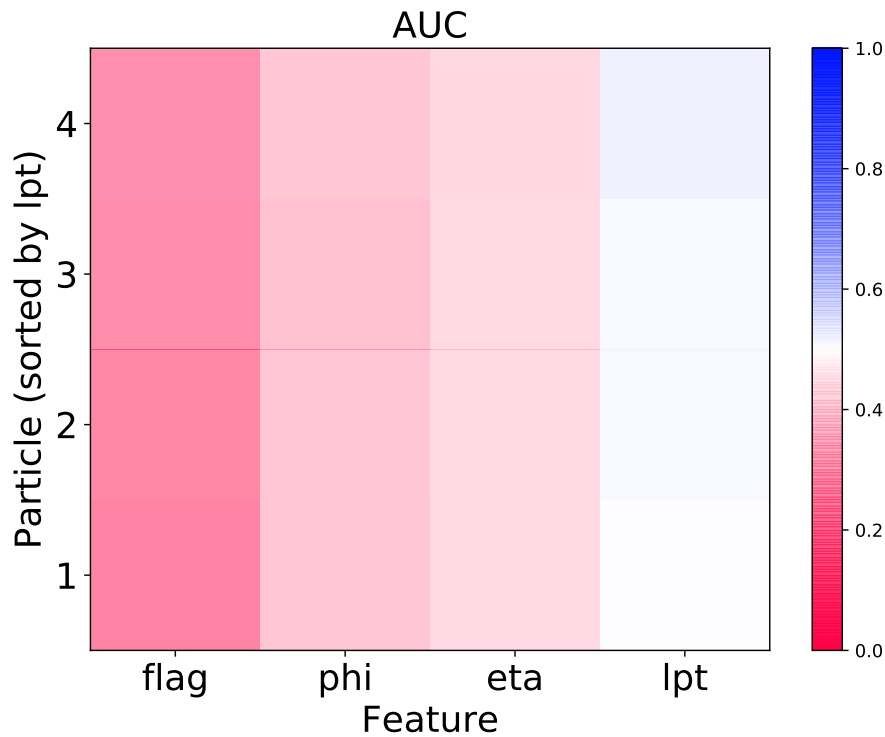


Figure 6.11: AUC feature map for a well normated network

But, as you also see, now most of the classification power lies now in flag, which is quite confusing: Something having no physical meaning being more useful than everything else. (Not to different compared to chapter 5.2.1). This we will explain in chapter 7.1.

<sup>73</sup>To be more precise, the difference between the first and the second particle is higher than the difference between the last two ones.

## 7 Mixed networks

Referenced in: [1] [4.4.1] [4.6.4] [5] [5.1.5] [5.2.2] [6.1.2] [9] [C.4.3] [D.1] [D.5.1] [E.3]

### 7.1 Oneoff networks

Referenced in: [3.3.2] [4.6.4] [6.1.2] [6.2.3] [7.2] [9] [C.3.2] [D.5.1]

Consider the following feature map of a well normalized network in figure 7.1.

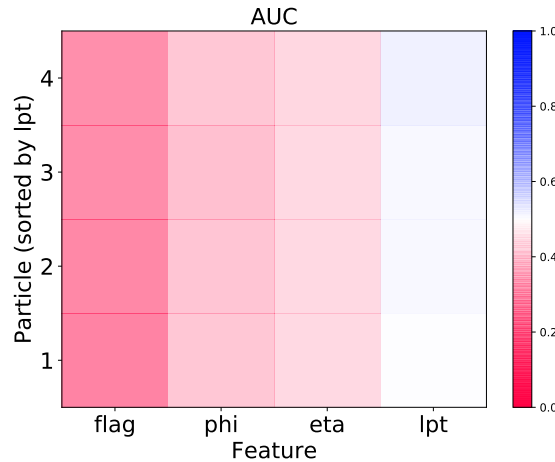


Figure 7.1: AUC Feature map for an on top trained autoencoder, using a good normalization

You see, that most of the decision power is in the first feature, but the first feature, *flag* is basically just one<sup>74</sup>. This might seem a bit counterintuitive or unphysical at first, how can a variable without any physical meaning be a better separator than those variables with physical meaning: To explain this, we need to take a bit more close look at what the network is doing: First, just because the output is has no physical meaning, this does not mean, that no physical variables are used in its calculation. In fact, before this we always just assumed that there is one parameter in the latent space, that is learned to be just a one from the input space<sup>75</sup>, but this distribution of decision power implies that this is not the case: If there would be a constant feature in the compression space, the constant output would be a trivial copy of this constant and thus have no physical meaning. More likely is the following: The network is able to reconstruct an 1 from all the other parameters. This makes sense, as we got this AUC distribution by changing the normalization in a way that made trivial ones in the input space much less likely<sup>76</sup>, and it also explains how an unphysical output can be physically useful: Since they are utilizing physical inputs, the resulting constant has to be a function of the inputs. And when you change the inputs, the constant is also changed and this change we can use to differentiate signal and background events. And since this quality is better than every other autoencoder decision quality (0.3 here, or 0.25 using only *flag*), it might be useful to use this further: If apparently nonphysical outputs can be at least as good as physical outputs, why not just use outputs that are nonphysical (Outputs that are one). This is what we call oneoff

<sup>74</sup>Flag is 1 as long as the current event does not contain fewer particles than the network demands, and since this is a network with only 4 nodes, and there are very few jets with only 3 particles or even less, saying *flag* is a constant ( $flag = 1$ ) is a quite good approximation.

<sup>75</sup>This is a bit of a simplification, most importantly it would be untestable, since instead of learning a constant, the network could learn a constant as a function of multiple parameters (for a simple example consider  $x_2 = x_1 + 1$ , both variables are not constant, making it harder to find this, but still  $x_2$  has no additional information with respect to  $x_1$ , and there is a one learned as  $-x_1 + x_2$ ).

<sup>76</sup>Since we stopped dividing by  $\max(|x|)$  and started dividing by  $(\max(|x|) + \text{mean}(|x|))/2$ , it is no longer the case that there is either a  $-1$  or a  $1$  in each feature.

networks<sup>77</sup> As shown before (see chapter 5.2.1), complexity is to a big part just width. You may be able to solve this by normalization, but this removes information, and oneoff networks would not require this<sup>78 79</sup>. Also there might be a certain kind of complexity benefit, since the whole network is made to just minimize one distance<sup>80</sup> that is always the same, instead of optimizing some feature that might be useful for some events, but useless while considering other events, in which this feature plays a less important role. This should result in the network being able to learn more complicated functions.

We justify this idea mathematically in appendices E.3 and E.4

### 7.1.1 Oneoff quality

A simple dense network with just an output that should be one, sadly does not work well in jets. First: the loss can go to basically zero ( $1/1000000000000$ ), which is a bit unphysical, since the loss as a distance to one, is basically the variance of the used feature, and you would not expect there to be any physically significant feature of this accuracy in 4 particles<sup>81</sup>. So there are features that are more trivial to learn, and make any decision process meaningless. And it is not necessarily trivial to find those, there might be those features that are just input variables with value 1 (for example an input that would be set to flag), but not all of them are that easy to find.<sup>82</sup> This makes training an oneoff network is a bit like outsmarting your algorithm. One thing that we found quite useful for this, is letting the network not only learn an 1 on the data that you are interested in, but also a 0 on other random data.<sup>83</sup> When we use relu<sup>84</sup>, learning values to be zero, means learning them just to be negative, and is thus way easier. This can demand that the network does not fixate on trivial features in the networksetup and preprocessing<sup>85</sup>. A simple oneoff network reaches usually an AUC of at best 0.6 for the task of finding top jets, which is not too impressive. But if you look at the classification power as a function of the training epoch, you see that this only is so bad, since those AUC scores are way better at earlier epochs (see figure 7.2).

<sup>77</sup>Since the distance off 1 is the deciding quality indicator and it is a OneClass algorithm.

<sup>78</sup>Since their output, 1, is obviously automatically normalized.

<sup>79</sup>Also in practice it seems to be still a good idea to normate also oneoff networks, this might be because this normalization also lets features the oneoff network focusses on to be more similar and thus easier to combine, or because similar sized inputs are easier to train on.

<sup>80</sup>Actually, in practice it seems to simplify the training, if you don't use only one output, but multiple ones, that all are compared to 1 and which mean is used. This results in very high correlations in the outputs, but seems to help in the convergence of the network.

<sup>81</sup>Especially, since the lowest difference there can be in the used float32 implementation is bigger than  $1/100000000$  and thus, since the final loss is the mean of each loss, this would mean, that at least a fraction of 0.9999 of the events reproduce exactly 1.

<sup>82</sup>A notable example might be the preprocessing of  $lp_T$ . As descibed in chapter 3.2, we used a preprocessing similar to that of ParticleNet:  $x = \log(p_{Tjet}/p_T)$ , but this means (because of the implementation), that a sum over  $e^{-x}$  is always 1. This might also be a good time to talk about functions in this kind of networks. Since we have to forbid any biases (a bias would just result in the network learning a zero and adding a one as bias), the usual reason for a network to learn any function has to be modified a bit. Think about taylor approximations: A function like  $e^x$  could be written as  $1 + x + O(x^2)$  (with as many terms as the networks needs), but for a network to learn 1, the input of  $e^x$  would then be set to zero, the network output would be one and it would basically be the same as adding a constant bias. But adding a bias is not allowed, and thus the network can not learn  $e^x$ . That beeing said, the network can learn  $e^x - 1 = x + O(x^2)$ , and, when  $\sum(e^{-x_i}, i) = 1$  then is  $\sum(-1 + e^{-x_i}) = -3$  for 4 nodes, and thus the network can learn this, without having learned anything physically useful.

<sup>83</sup>We choose here random events with the same mean and standart deviation in each feature, as the original data, that still goes through the same preprocessing.

<sup>84</sup>A relu activation can be defined as  $x + |x|$ . See Appendix B.2.3 for why this is useful.

<sup>85</sup>Later on, in chapter 7.2, this is no longer needed, and just complicates the training.

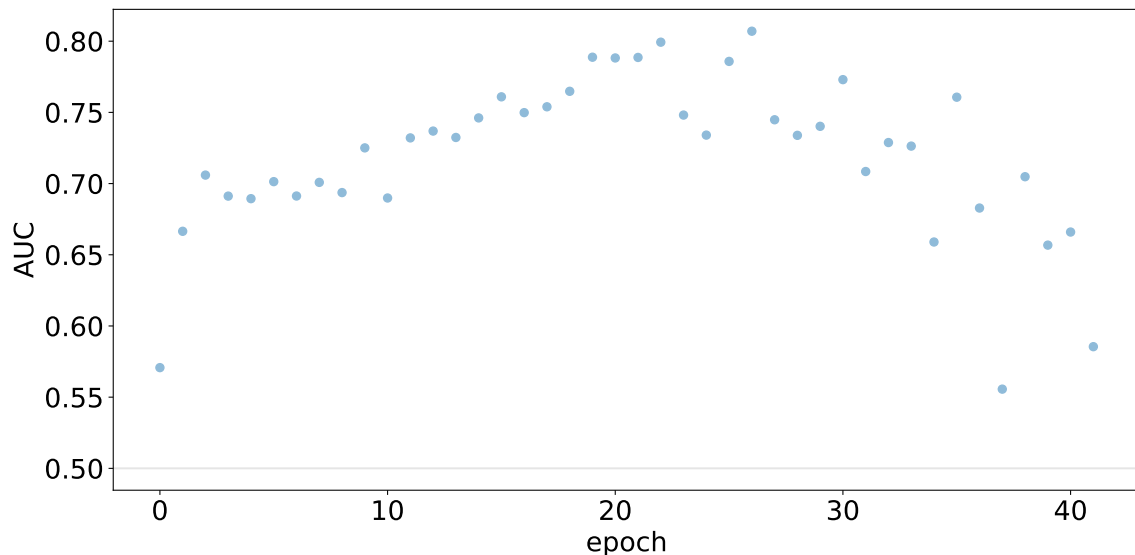


Figure 7.2: AUC score as a function of the epoch, trained on QCD, here for a graph oneoff network. Graph oneoffs are not used anymore in the following, but since they show the same relation as a dense oneoff network much cleaner, we use this curve here. As you see, the relation shows a maximum before the training ends.

Sadly, this observation is not really useful, since stopping the training at the optimal epoch would not be unsupervised. But it is still quite interesting, since it shows, that there is some potential in those kind of networks, which is just not utilised good enough. Another problem is again invertibility: It is possible to create an invertible oneoff network, but it is not trivially given. This becomes easier, when you use a lot of trainable parameters. To do this, a graph network is less useful, than just a simple dense network.

Even though they are not yet applicable here, we show in appendix E.4.1 that oneoff networks are very useful for finding anomalies in other datasets. This allows us also to suggest that combining multiple oneoff retrains can increase the classification power even further. We also show that you can use oneoff networks to extract human readable information from physical events in appendix E.4.2.

## 7.2 Latent space oneoff learning

Referenced in: [7.1.1]

The main problem of autoencoder might be the fact that its loss function is not necessarily the best possible separator(see chapters 4.5 and 5), while the problem of oneoff networks seem to be that they focus on useless information, which keeps them from reaching their optimal classification power(see chapter 7.1), but maybe combining both methods could solve both problems: You train an autoencoder to convert the input space into the latent space, to be able to run an oneoff algorithm on this compressed space<sup>86</sup>. This means that the separation function is now quite good (as suggested in appendix E.4), and the autoencoder can filter out those trivial inputs hurting the oneoff training curve.

This idea of combining networks is not exactly new(see for example [39]) and it also is harder to train, since we now have two independent networks: Something that improves the first network might hurt the second, but in practice this works quite well. It is not yet clear if you want to train your autoencoder on the background data or on both the signal and the background. Here we train on background data, since every bit of higher inaccuracy that might

<sup>86</sup>We also tried alternative algorithms, but oneoff networks result in the best results, see for this appendix E.2 and E.1.

be reached by giving the original autoencoder unknown data, will help the following algorithm, but the effect of changing this is tiny anyways. Also choosing the most unsupervised algorithm is not so easy: Defining a set with absolutely no anomalies is not completely unsupervised, but defining a set that is exactly half abnormal might be even worse: The anomalies we search are probably quite rare, and approximating this fraction as 0 seems to be more realistic than approximating it as 0.5.

### 7.3 A final classifier

Referenced in: [A.5] [C.4.3]

With the same setup as before (see chapter 4.4) and normalization as well as after training 25 oneoff networks on each latent space we gain the final classifier for this thesis

#### 7.3.1 Trained on QCD

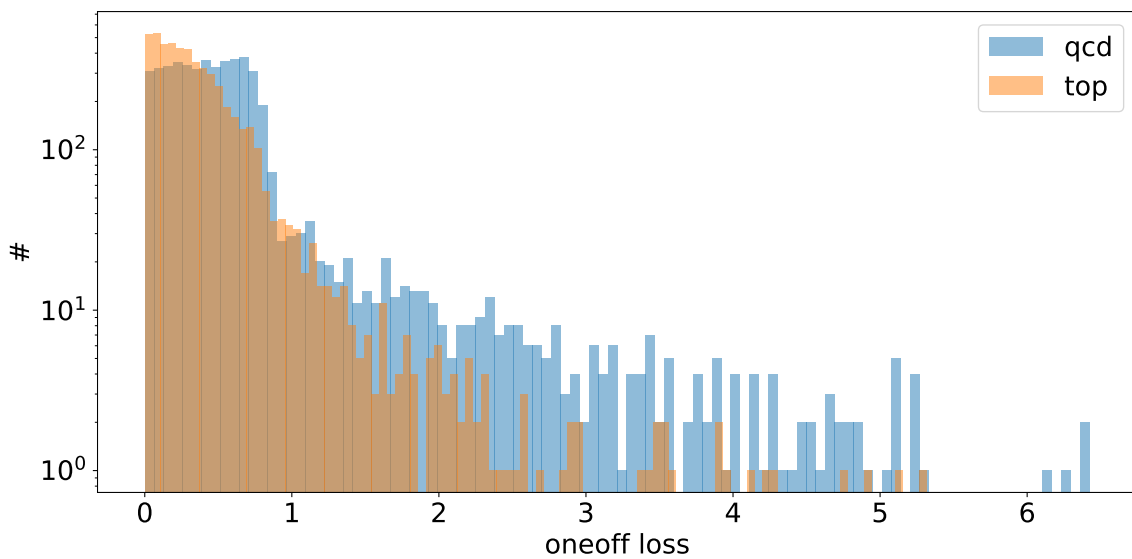


Figure 7.3: Oneoff loss distribution for a network trained on qcd jets

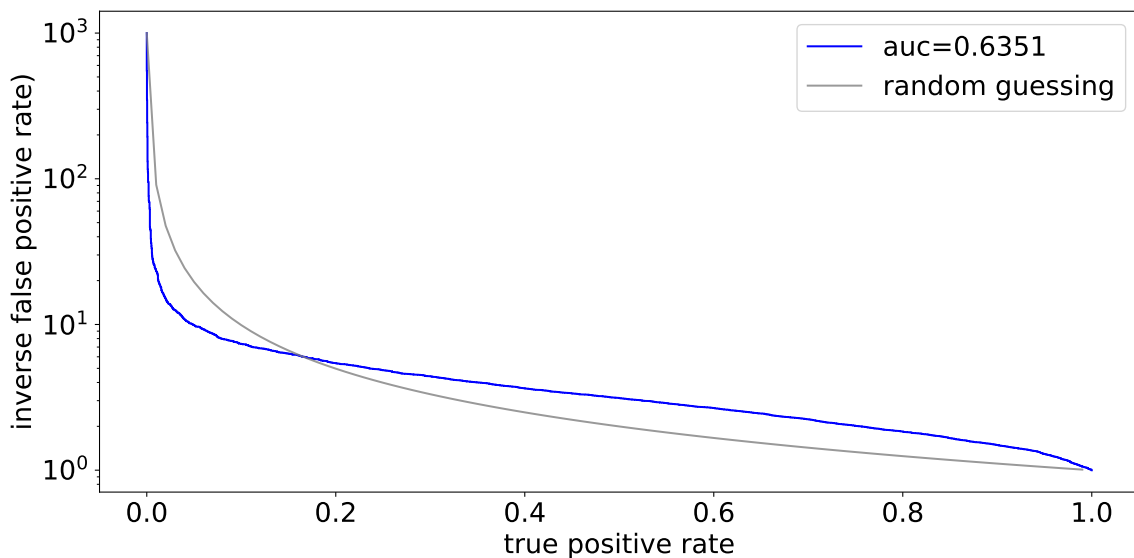


Figure 7.4: Oneoff Roc curve for a network trained on qcd jets

In figures 7.3 and 7.4 you see AUCs worse than in chapter 4, but consistently better than by using just normalization.

Interestingly this also helps the reconstruction quality (see figures 7.5 and 7.6).

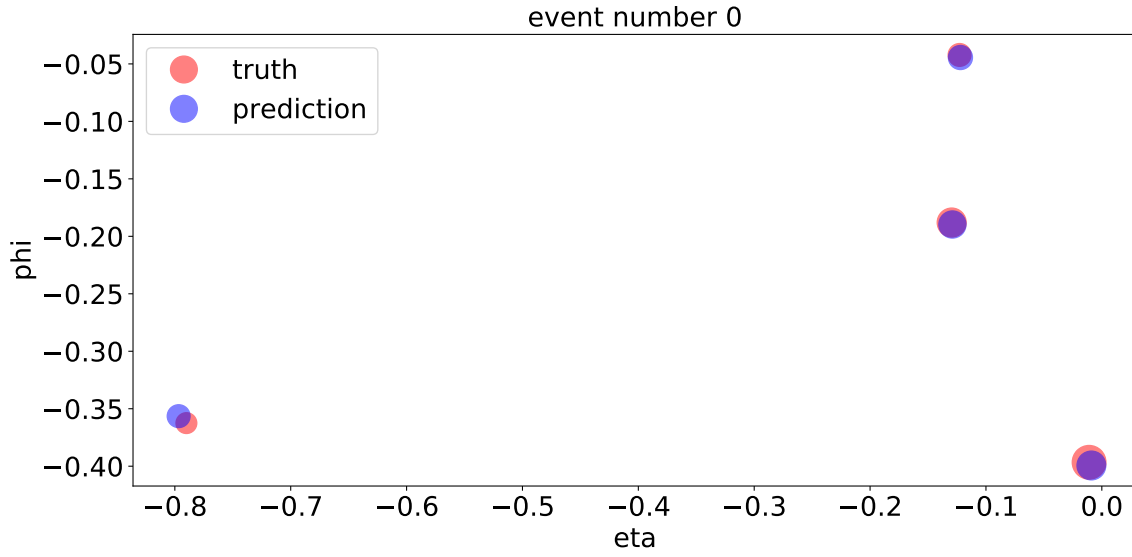


Figure 7.5: Angular reconstruction images for a normalized network trained on QCD

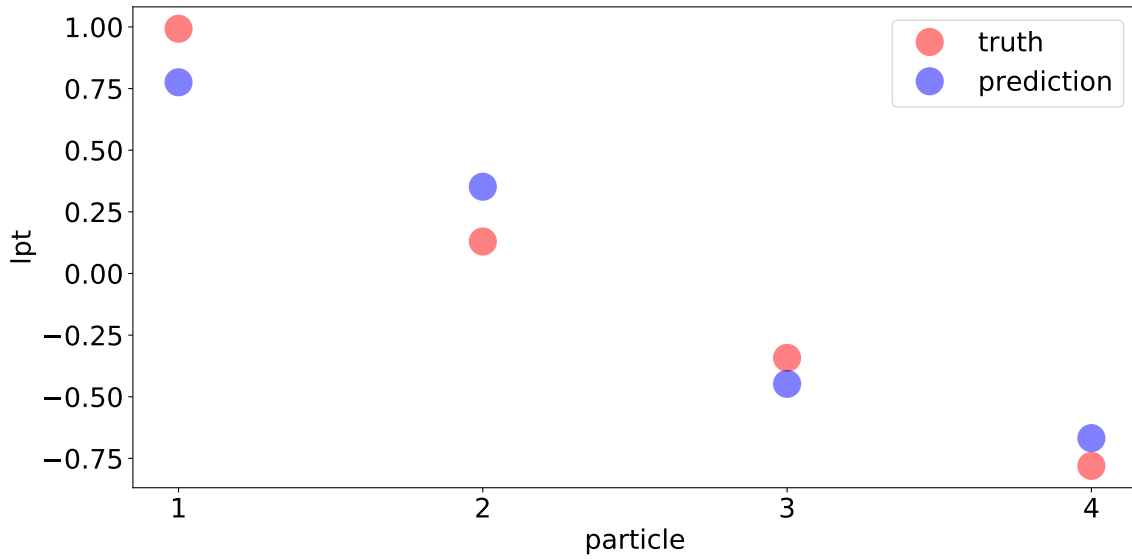


Figure 7.6: Momentum reconstruction images for a normalized network trained on QCD

### 7.3.2 Trained on top

Trained on top in figures 7.7 and 7.8, this improves quite a lot.

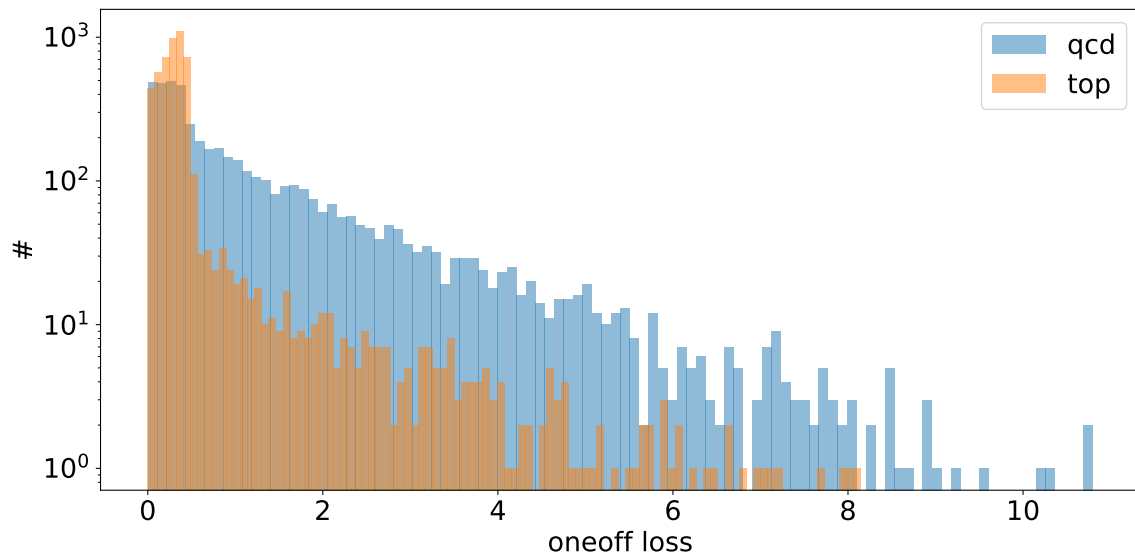


Figure 7.7: Oneoff loss distribution for a network trained on top jets

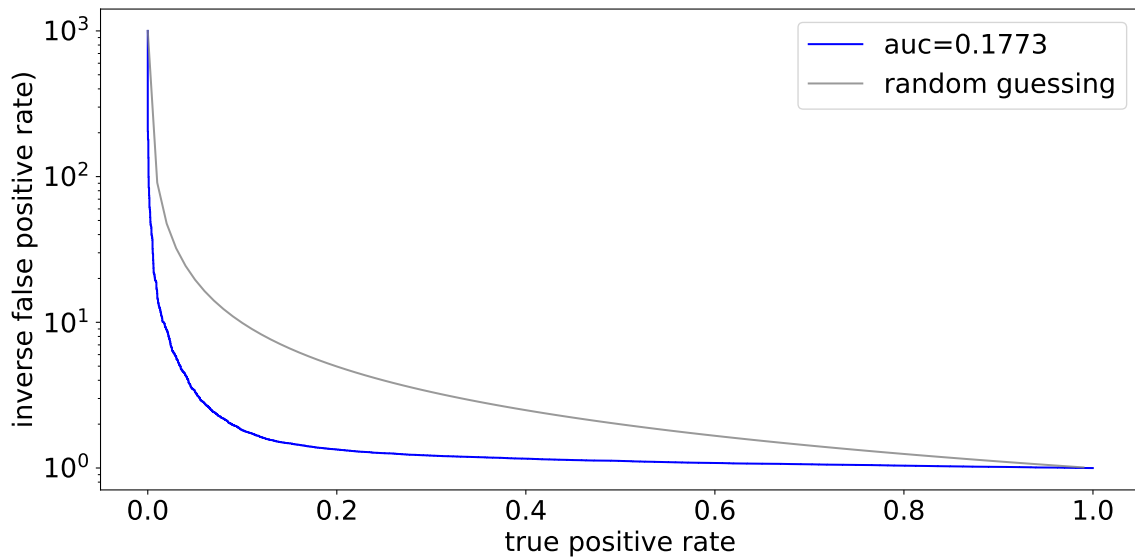


Figure 7.8: ROC curve for a network trained on top jets

Also, here the reconstruction quality is quite good here, as figures 7.9 and 7.10 show.

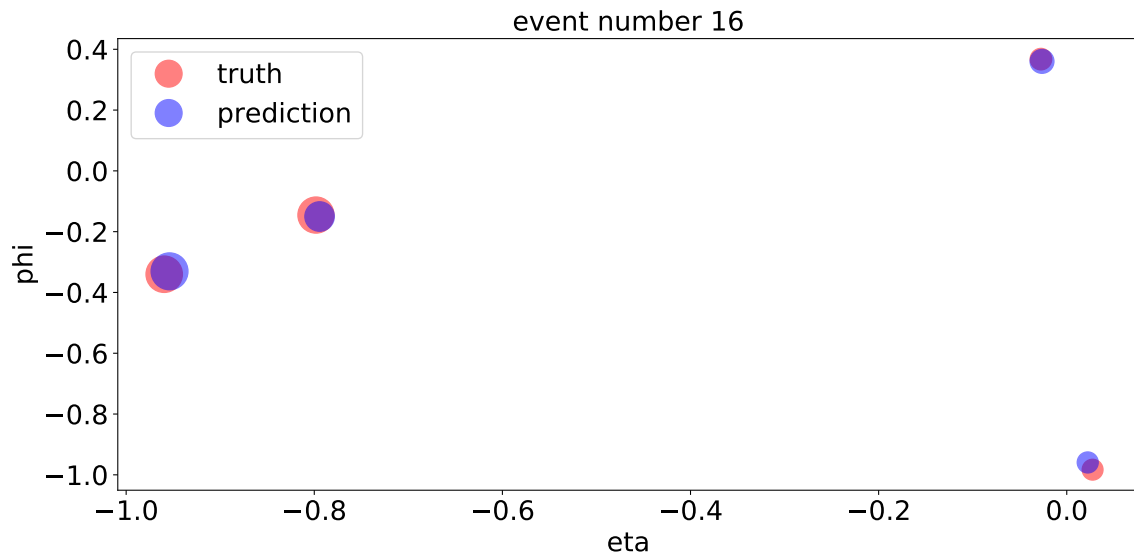


Figure 7.9: Angular reconstruction images for a normalized network trained on top jets

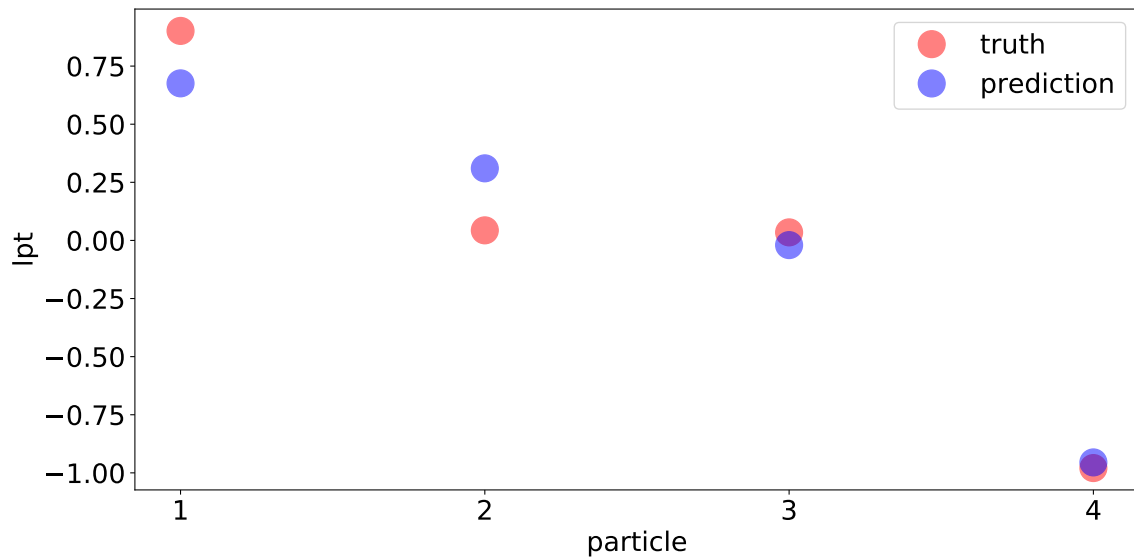


Figure 7.10: Momentum reconstruction images for a normalized network trained on top jets

## 7.4 Scaling with oneoff networks

### 7.4.1 Scaling in batches

The batches considered in chapter 6.2.1 are now all invertible, as figure 7.11 shows.

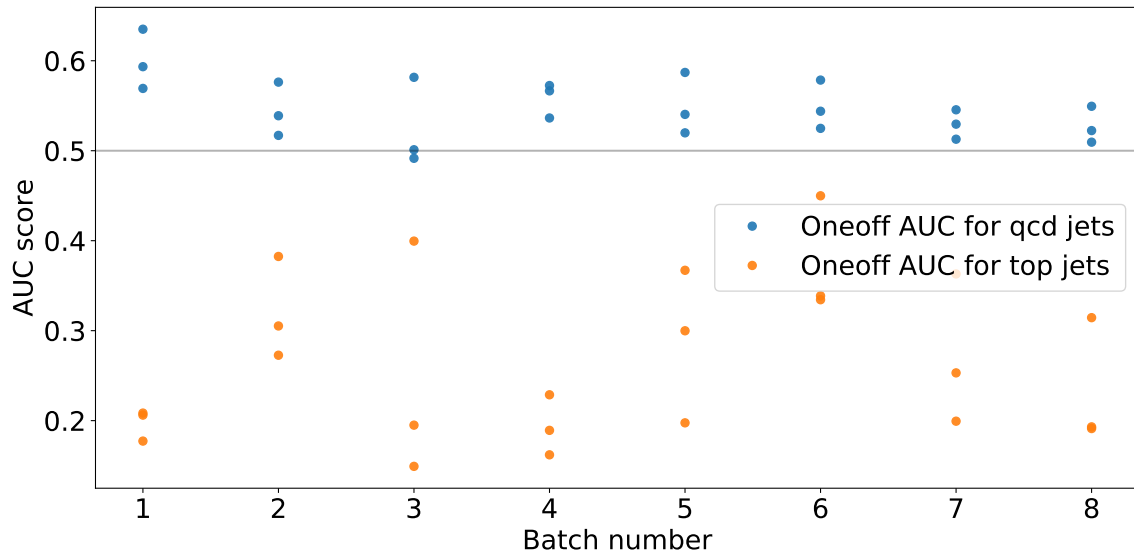


Figure 7.11: Invertibility of batches in oneoff networks

Here you see a much more interesting relation compared to before. The variance of each AUC score grows with the batch index, which is expected as they are more random, but some networks actually beat the AUC score of the first batch (batch 3 has an event below 0.15). This is a result of the number of particles in each jet becoming a feature at some point. You see this, by noticing that the relation between AUC and batch number is not linear: The AUC's for the second batch might even be some of the worst, even though they should have the second most information next the first batch. Sadly this nonlinear relation makes combining batches hard.

### 7.4.2 Scaling without batches

Referenced in: [4.8.2] [5.1.3] [5.1.5] [6.2.2] [8.1] [B.5]

From a technical standpoint, bigger networks don't train as well, since their loss becomes NAN at some point. This we can fix for now, by giving up two things: We cannot use a learnable graph anymore, and we train on fewer data. Using a fixed fully connected graph is usually not a good idea, as it seems to slow down the training, but this also removes a lot of NANs<sup>87</sup>. Using less data should not matter to much, since for 4 nodes appendix B.5 shows that it does not change anything to reduce the number of training samples to 5000 QCD jets. This removes fewer NANs, but has the added effect of accelerating the training a bit. It is also useful to use the normalization from chapter 6.1, as not using it seems to produce much more NANs.

We train with a batch size of 100 and a learning rate of 0.003 for at least 500 Epochs and afterwards with patience of 100 Epochs an autoencoder compressing 16 nodes twice by a factor of 4 into one node with dimension 36 building our latent space. Also, between each compression step, there are 3 graph update steps. This results in the training history shown in figure 7.12. This training took more than 58 hours training on a cpu<sup>88 89</sup>.

<sup>87</sup>It is always sadly possible for a network to NAN, which makes debugging harder. Removing the topK algorithm seems to make them appear mostly earlier, resulting in NANs appearing either in the first epochs or not at all, and thus allowing us to not waste any time training failing networks.

<sup>88</sup>Training on a gpu would accelerate this quite a lot. We expect a factor between 3 and 5, but since this still would not make gpus possible in our computation quota, we use cpus.

<sup>89</sup>Training a 4 node network takes about 3 hours, so the quadratic scaling expected for the graph update layer expects a training time of about 48 hours. This difference is most prominent, when you consider that we need train 4 node networks on 50000 training jets. You might explain it by our model being too easy (we have more than just graph update layers) or by our implementation not being as fast as possible.

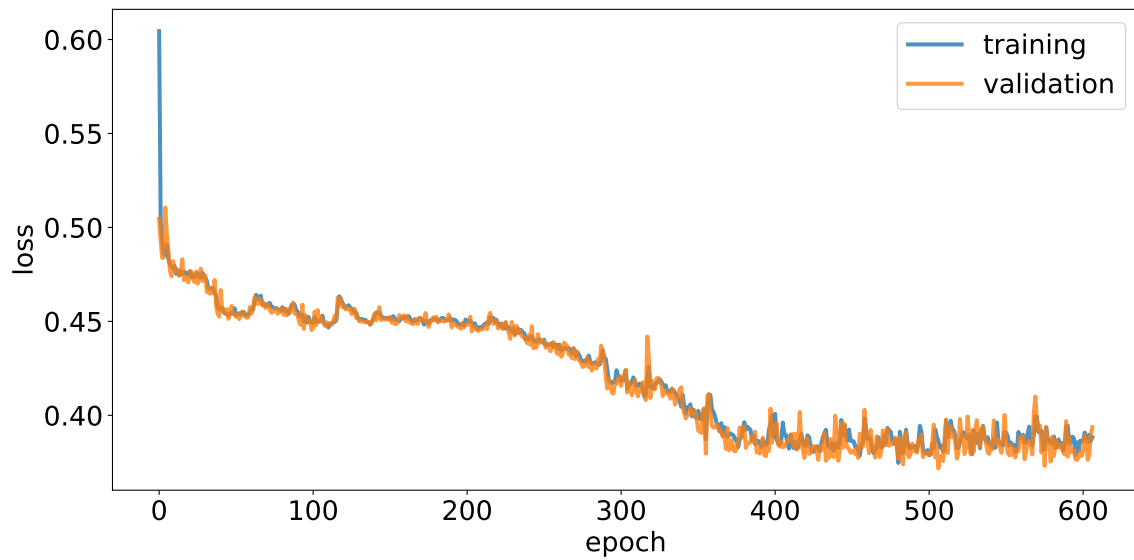


Figure 7.12: Training history for a 16 node network trained on QCD

More importantly, the reconstruction works fairly well, as figures 7.13 and 7.14 show.

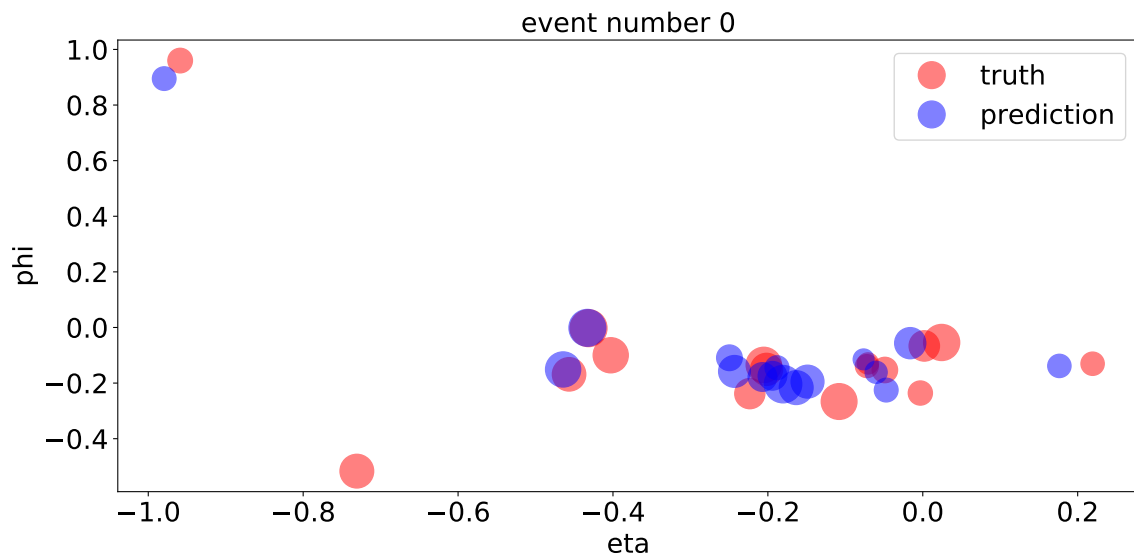


Figure 7.13: Angular reconstruction for a 16 node network trained on QCD

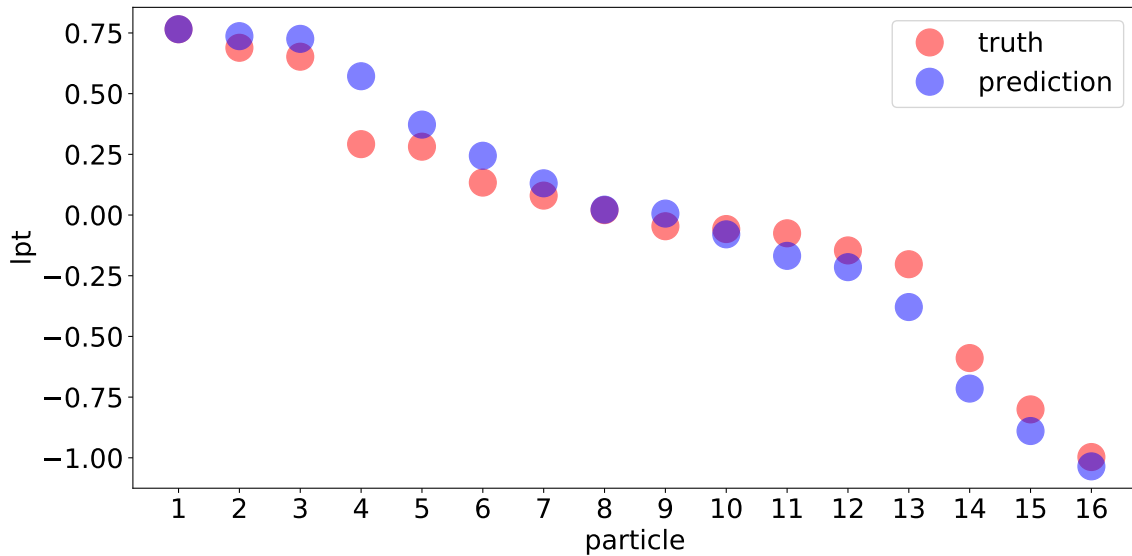


Figure 7.14: Momentum reconstruction for a 16 node network trained on QCD

Without oneoff networks, the classification quality is also better than our best AUC on 4 nodes (0.635). We move figures G.2 and G.3 to the appendix, but they show an AUC value of about 0.7.

Using oneoff network this AUC falls to 0.55 (see figure G.4). This might seem like oneoff networks are not as good as we assumed before, but this is actually not the case. Consider the same training done now on top jets. The training history (figure G.5) and the reconstructions (figures G.6 and G.7) are very similarly good, which is why you find them in the appendix. The problem lies in the ROC curves (figures G.8 and G.9). They reach an AUC score of 0.64, and we thus would have networks that are not invertible. We think, that these networks are not invertible (even though we use normalization) because the normalization has a much lower effect: On 4 nodes, removing two values means removing 1/2 of all values, on 16 nodes this only means removing 1/8 of all values. So, since the trivial difference is contained in each particle, and slightly differently for each of the particles, removing 2 values might remove some width, but in the substructure there is still enough contained for the network to only use a triviality. So we need to use our improved way of handling trivialities: Using oneoff networks here results in an AUC score of 0.48 (see figure G.10) and at least making this network invertible.

These terrible AUC scores show that simply solving the computation challenges of more node networks is not enough: We think that by adding nodes that are more and more random, the autoencoder focuses on reconstructing them more than about the first nodes. But since in these initial nodes most of the classification power is contained, this just weakens the classifier. So you would need to also keep the focus of the network right. One way of doing this, would be weighting your loss function, but our experiments with losses that are functions of the node index or the transverse momentum only worsened the reconstruction quality (see chapter 5.1.5).

## 8 Applying this model to other datasets

Referenced in: [1] [4.6.4] [5.2.1] [5.2.2] [9] [9.1] [D.5.1]

We might be able to supervised separate probably any kind of data, but as shown in the previous chapters, if we remove the labels, this exercise becomes a lot harder. Here this problem is usually stated as follows: Given a set of data points, can we write an algorithm to detect a second set of data points. The only difference to the supervised case is the fact that we cannot look at the anomaly set in training. And information from the validation dataset can leak into the model setup[13]. This is an effect, that is usually solved by introducing test data. Data that is only used once at the end of your analysis: If your network works worse on this data, then your setup contains information about your validation data and is no longer as general. We want to use this chapter to introduce our test data. The difference here is, that we cannot simply use some part of our validation set: It are not the events of the validation set that are leaking(as shown in 4.7 our networks do not overfit), but the specifics of our anomalies. So as test set, we have to use completely different anomalies<sup>90</sup>.

You could see this, as changing our initial task: Instead of finding one specific anomaly, we now want to find every other anomaly. One should notice the huge difference in complexity of this task: Defining every alternative dataset as signals is not solved by looking at any attribute to differentiate datasets: There will always be another dataset, that is entirely the same as the background set, if you are looking at this attribute only. And there will even be a dataset, that has an attribute that looks more than the background than the actual background<sup>91</sup>. You could say, that finding all alternative datasets, is more about defining your background, than about finding differences. But as even oneoff networks, that are designed to define your background datasets, in theory are expected to be able to be trapped by some features(see appendix E.4), the only way to truly evaluate an algorithm, is experimentally. And since we cannot generate all alternative datasets, we have to work with comparing specific two datasets to each other. But we can at least give some sense of generality to the networks, by looking at different kinds of datasets.

### 8.1 Ligth dark matter

Referenced in: [8.2.2] [8.3] [9] [9.2] [E.5]

This set of data points is generated by Thorben Finke and consists out of jets of transverse momentum between  $150 \cdot \text{GeV}$  and  $270 \cdot \text{GeV}$  of either QCD jets, or those initiated by a dark matter candidate sugested in [11] (ldm data). This dataset implies an unsupervised classification task that is way more difficult than the usual top tagging, and as we will see, even more complicated than the other datasets that we test our algorithm on here. The first thing that makes this dataset so much more complicated is the angular distribution: while you can use this distribution to differentiate top jets from their QCD counterparts alone quite well (see chapter 5.2.1), here both angular distributions (figure 8.1) are basically the same.

<sup>90</sup>Training on your anomalies to find your background can help, but even this can not really exclude that your data leaks into your model(If your signal and background differ completely in one parameter, optimizing either would result in a network only focussing on this parameter, and thus not beeing very general), so the only real way might be to train on as much datasets as possible, and demand that all work.

<sup>91</sup>Looks the same as the background but with lower width.

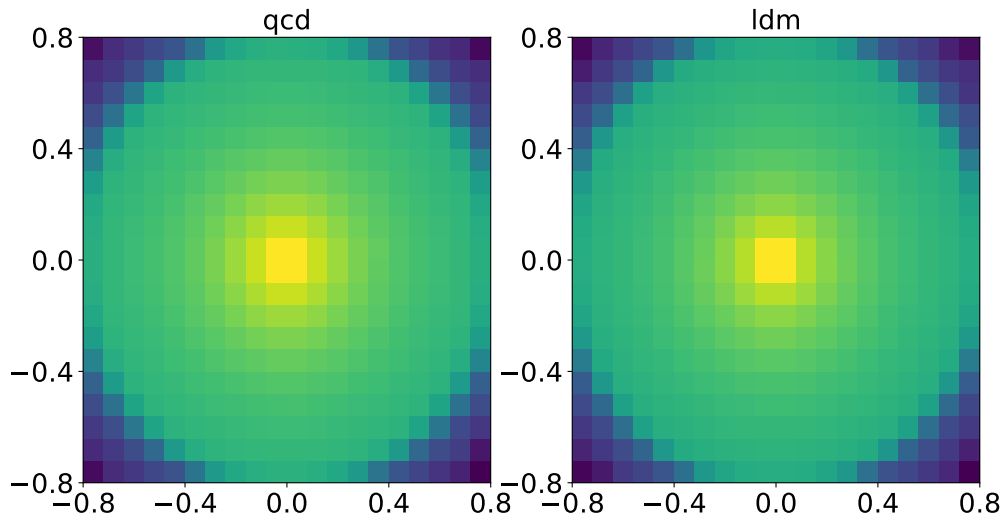


Figure 8.1: Angular distribution of ldm jets

Also the momentum distribution in figure 8.2 is not much better.

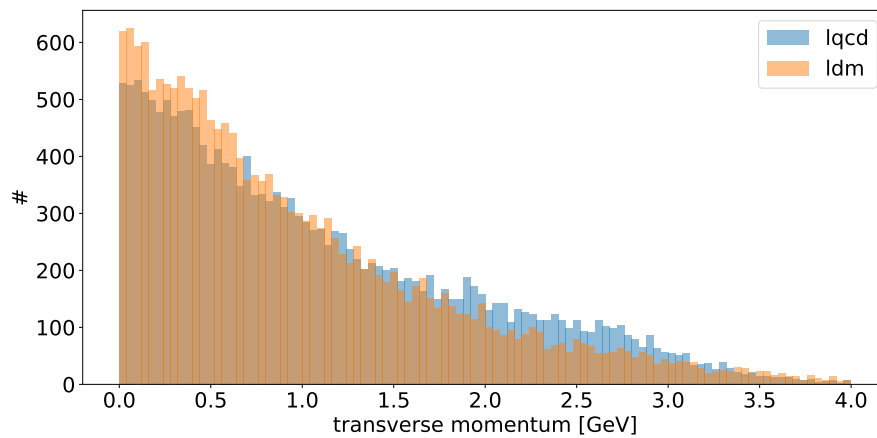


Figure 8.2: Momentum distribution of ldm vs lQCD jets

That beeing said, there is one easily understandable parameter that can be used to differentiate both datasets: The number of particles in the jet, which is shown in 8.3.

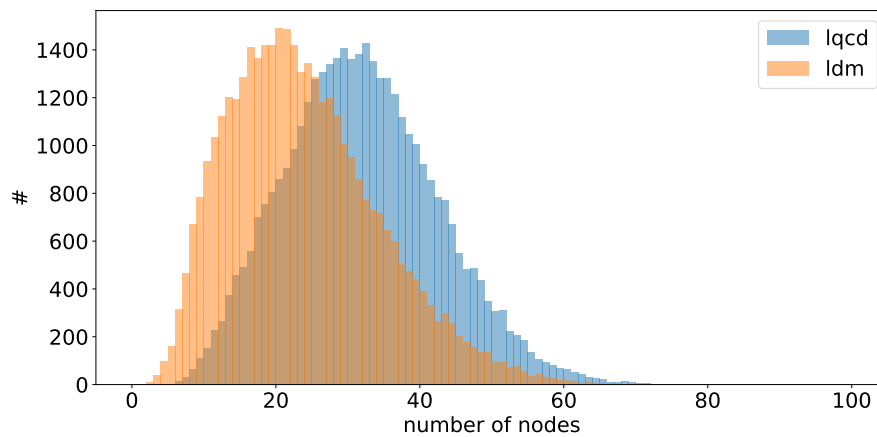


Figure 8.3: Number size distribution of ldm jets

Sadly, this parameter is not very useful because of two reasons:

- Our graph based networks, especially the ones we talk about in this chapter only have 4 particles. That means that the number of particles just does not enter the network at all<sup>92</sup>, and even though we can increase the number of particles that enter the network, this will result in less well-trained networks (see for example chapter 7.4.2), and at the end.
- We actually don't want to have a network that just focuses on the number of nodes, as this is a fairly weak way of differentiating jets, resulting only in  $O(1)$  AUC values in the best case. And even though this is way better than any classification score that we achieve in the following, focussing on only one parameter loses every sense of generality.

This means, that we train 4 node networks<sup>93</sup>, that hopefully find some sense of substructure, which will make it possible to differentiate those jets. Sadly this also means, that every result has to be fairly bad. There is a dataset with different substructure, but there is also another dataset with completely different angular distribution, and since we want our networks to find both, this also means, that having the same angular distribution has some effect making the network consider those sets as more probably the same datatype. There might be still some different substructure, but there are three effects here making the network weight its possible differences:

- Your first thought might be, that this relative uncertainty is not affected by the function complexity, and it is true, that if you got only two variables, their effect is completely unaffected by it, but there is a catch.
- You cannot assume each variable to be exactly known, and a slight variation in a complex formula can have a much bigger effect than in an easy formula (think of a momentum 4 vector representing an electron: the formula getting the energy from this 4 vector is much more stable under variations of the 4 momentum, than the formula getting its mass).
- Also, the number of neurons is finite, and you could argue, that so is the number of calculable variables. This means that the network has to choose favorites. And since every given feature can be weightedly added in a way that (at least for a tiny amount) improves the current relative uncertainty, choosing a complicated/expensive feature also means, not choosing multiple less complicated features.

This means, that there is a slight preference of oneoff networks, to choose easier features<sup>94</sup>, which means, that this is a really hard test for them, and the only thing that we can realistically demand here is invertibility: A Network, trained on light QCD jets, that thinks ldm jets are more complicated as well as the inverse.

---

<sup>92</sup>To be precise, there are  $O(10)$  jets with less than 4 particles in our dataset, so it actually is inputted, but only to a negligible amount.

<sup>93</sup>The low number of particles becomes a benefit, since we can be sure not to use the particle number.

<sup>94</sup>In general, this is actually a really good thing: Not only does it is statistically useful, but this also means, that oneoffs have a built in regulator, that prevents them from overfitting (at least to a degree), making them quite general.

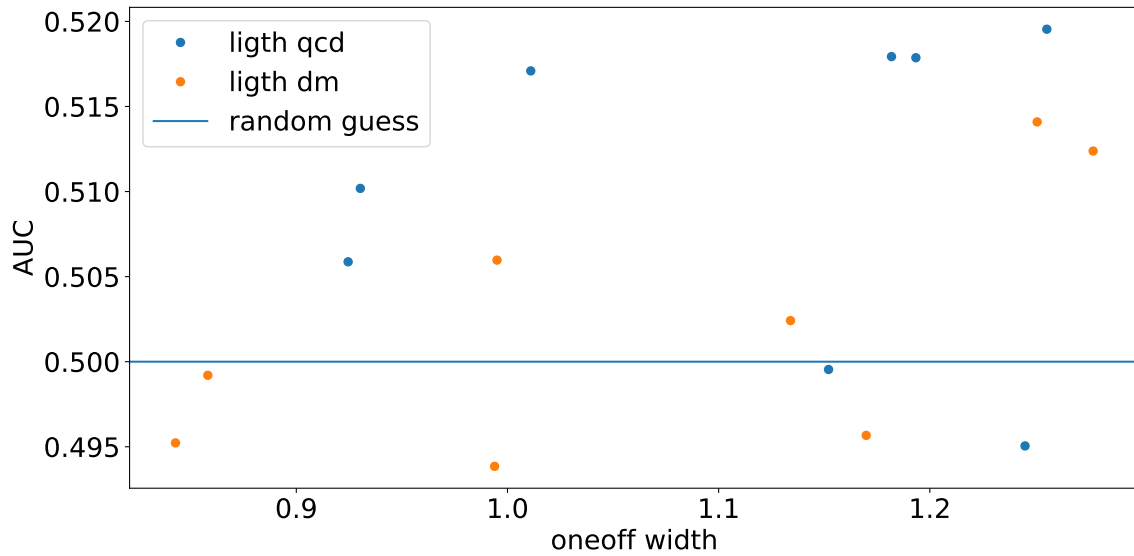


Figure 8.4: ldm jet invertibility

As you see in figure 8.4, this is not at all trivial, but when we consider the loss of the oneoff network, which is drawn on the x axis as quality, the best networks are actually invertible. And since this is still completely unsupervised, just using the feature quality of a network, we can say that we can generate invertible anomaly detection algorithms on this dataset. That being said, this is obviously not useful at all, as half a percentage in AUC does not help you finding differentiating new physics, but it is worth to note, that also top tagging after normalization looked very quite bad (see chapter 5.2.2), and seeing that there the classification quality improved a lot, we see no reason, why you could not improve and optimize this network to be drastically better. Especcially, since we did not run any hyperparameter optimization (except for the compression size, which is 1 bigger (at 10)<sup>95</sup>), and still only use 4 particles. This analysis is the reason, why we think that the oneoff width from chapter 4.6 is such a good way of evaluating a model.

## 8.2 Other datasets

### 8.2.1 Quark or gluon

Quark and gluon data, is generated by Madgraph[9], Pythia[50] and Delphes[41]. One set is generated as parton parton to gluon gluon collisions and another as parton parton to two partons without gluon collisions. Jets are used, if their transverse jet momentum is between 550 and 650 GeV. This data was used originally generated to see if a QCD trained classifier makes an easily accessible difference between quarks and gluons<sup>96</sup>, but even though this is seems not to be the case, we can still use this dataset to test our algorithm a bit further. Again we use 4 particle networks, with a compression size of 9 and only negligible hyperparameter optimization to reach quality of slightly above random.

<sup>95</sup>We think, that this higher compression size allows the network to understand more subfeatures.

<sup>96</sup>You could interpret this, as another form of complexity: while top jets are all the result of top quarks, with QCD jets there are multiple options, we though this could explain why QCD trained encoder are generally worse, but this seems not to be the case.

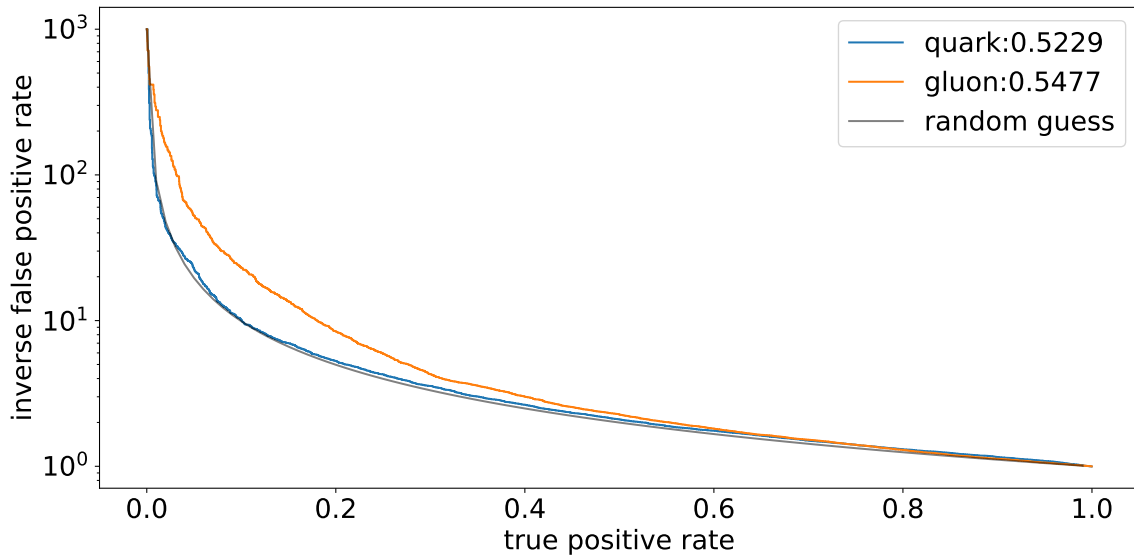


Figure 8.5: Oneoff ROC curve for quark gluon. Here both curves should be above the line representing random guesses, and as you see, this is the case, even though the quark line is very close to randomly guessing.

As you see in figure 8.5, these are invertible networks, and even though they are not very good ones, as described in the previous chapter 8.1 this does not really matter, since optimization has the potential to improve them quite a lot. Here [36] could be seen as a reference paper for this process, even though they use a supervised approach and high level input data on different transverse momentum ranges, their achieved AUC values below 0.9 suggest that this tagging job is a bit more complicated than the usual top tagging task. Chapter 8.3 will support this hypothesis.

### 8.2.2 Leptons

This dataset is not very physically useful, but more interesting from an anomaly detection standpoint: We again generate particle collisions using Madgraph, Pythia and Delphes, but instead of partons colliding into partons, we use leptons colliding and producing partons. For the first set, we use any combination of electrons and muons with arbitrary charge, and for the second one we only use tau leptons. We also use a fairly big transverse momentum range for the jet of  $20 \cdot \text{GeV}$  to  $5000 \cdot \text{GeV}$  to see if our algorithm is affected by this bigger range.

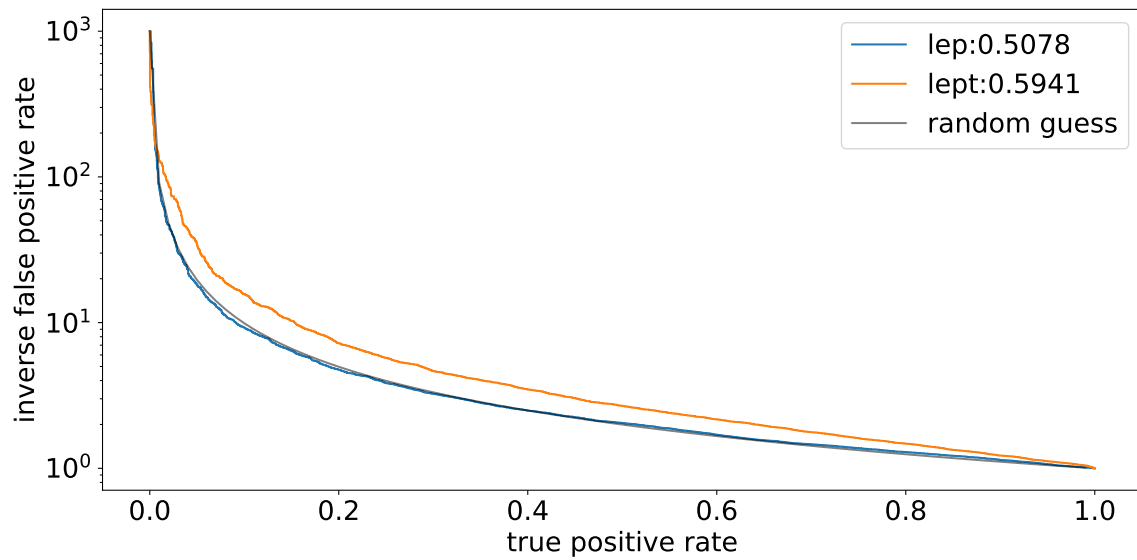


Figure 8.6: Oneoff ROC curve for lepton data. Again both curves should be above the random guessing line

Again you see a slight invertibility here (figure 8.6), helping to support the suggested generality.

### 8.3 Cross comparisons

Referenced in: [8.2.2] [E.5]

At the beginning of this chapter, we called anomaly detection the task of finding everything that is not similar to the trained on class. And even though we tried to evaluate this task, by showing the invertibility on a multitude of datasets, we slowly are out of particles to test it on<sup>97</sup>. That being said, one thing we did not yet do, is to mix the datasets. You might question how useful this is from a physical standpoint, as there will probably never be a situation, in which you want to find leptons, only knowing gluons. The point is that new physics could have a nearly arbitrary form, and even though we will never live in a world in which we only know about gluons, finding data that does not look like gluons is very useful. We think that these experiments introduce thrust in the algorithm used, as chapter 5 clearly shows, that invertibility and feature triviality can be linked. And since training unsupervised does mean that we don't have to train new anomaly detection models, there is no reason not to compare those jets.

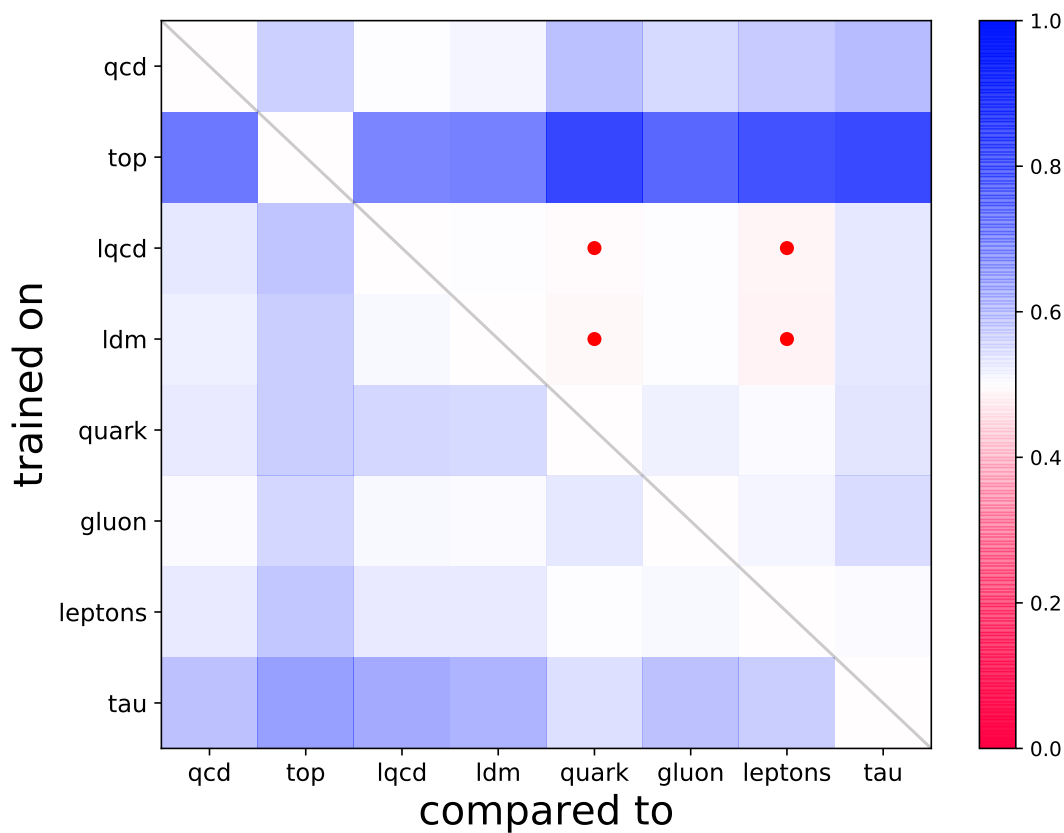


Figure 8.7: Comparing each dataset to each other dataset, using oneoff networks. We use red points to mark values below 0.5

In figure 8.7 you see, there are only few spots that are not invertible (we changed the meaning of each AUC value, in such a way, that each slot should be deeply blue in the best case). For simplicity, we mark the noninvertible networks with red dots, but this still does not allow to see every value, so here are those comparisons again as a table.

As you see, everything is either invertible, or at least very close. Furthermore, only 4 noninvertible comparisons exist, and are always less than 2.5% worse than random guessing, and are trained on ligh dark matter or ligh QCD data, which as explained in 8.1 is hard to differentiate<sup>98</sup>.

<sup>97</sup>Especially since the initial toptagging dataset already contained the whole of QCD.

<sup>98</sup>It seems a bit weird, that ligh QCD jets are more similar to ligh dark matter data, than to QCD of higher

	QCD	top	lQCD	ldm	quark	gluon	lepton	tau
QCD	0.5	0.5958	0.5048	0.5181	0.6333	0.5792	0.606	0.644
top	0.7893	0.5	0.7589	0.7656	0.8905	0.8248	0.8672	0.8879
lQCD	0.5476	0.6219	0.5	0.5022	0.4936	0.5037	0.4822	0.5471
ldm	0.5339	0.6043	0.5134	0.5	0.4863	0.5	0.4768	0.5456
quark	0.5429	0.6019	0.5828	0.5773	0.5	0.5251	0.5063	0.5535
gluon	0.5063	0.5831	0.514	0.509	0.5455	0.5	0.5167	0.5744
lep	0.5448	0.6173	0.5448	0.5403	0.504	0.512	0.5	0.5089
lept	0.6254	0.7018	0.6753	0.6595	0.5647	0.6283	0.6023	0.5

Table 8.1: Cross invertibility auc scores trained on row to be compared to column

Also note, that the rows and columns that are related to top jets are clearly visible: It seems to be a much easier task to differentiate top jets, than every other dataset, even when we use normalized networks. This suggests that only using top jets as anomalies artificially inflates your performance.

---

energy, especially since every value is normalized quite thoroughly (chapter 6.1). This still does not mean, that there is something wrong with the data, as the normalization has no effect on the number of particles, but just on the size of each value. And since higher energy jets can decay differently, this might explain why ligh QCD and ligh dark matter jets look so similar.

## 9 Conclusion

Referenced in: [1]

In this thesis, we were able to implement a working graph autoencoder. Consisting out of multiple different encoder and decoder algorithms we optimized it to work well on jets. Since our code (grapa) is very modular(see the documentation at <https://grapa.readthedocs.io/en/latest/>), you should also be able to use it on other tasks. We provide some examples for this in the documentation (see appendix F). For jets, we created a custom loss function to make our Autoencoder work more image like (see chapter 4.5) and show that our autoencoder it is not only able to reconstruct jets (see 4.7), but also that it works well as a classifier: On only 4 particles it already reaches a good AUC value for the task of unsupervised top tagging of over 0.85 (see 4.8). Optimizing this, using among other things a method we call c addition (see 5.1.4), we were able to extract from those networks an easy feature allowing for a comparison with an AUC of over 0.9155. This result might be comparable to our literature comparisons (see 5.2.1), but since this feature is fairly trivial, this also means that our networks, and this feature especially, is not useful for finding new physics at all, as it only uses a trivial difference in the angular radius between our background and signal dataset.

So we were in a situation, in which we cannot assume this difference to be there in new physics models, and in which we can also show, that our networks coincidently tends to use this feature regardless of the data we consider as background. This means, that if we ever want to be able to find new physics events, we have to make a network ignore this trivial difference. Chapter 6 tries to remove this trivial difference from jets using a creative normalization. This hurts the classification power, since we removed information from each jet. But this also creates a network that, if we consider our usual signal top jets as background, is also able to find our usual background QCD jets as anomalous, which suggests that it is no longer using a trivial feature. Considering QCD jets as anomalous it reaches an AUC of about 0.623 on 4 particles. By changing the normalization, we can produce a classifier that seems not directly to use physical features, while still improving the networks classification quality to 0.75. Chapter 7 uses this apparent inconsistency to create what we call oneoff networks: Networks that are trained to output a constant and use differences to this constant to find anomalies(see 7.1). We mold these oneoff networks into a tool being a better nontrivial classifier on the latent space of our autoencoder, improving the previous task of anomalous QCD jets to an AUC of 0.823. And while the more classical task of finding top jets as anomalies is a bit worse (0.635), this should still be more useful than the trivial AUC of 0.9155 above, since it might be able to find anomalous events of any form. To test this, and to make sure that our networks are not again only learning a trivial feature, chapter 8 uses other datasets to test how generally oneoff networks can find anomalies. From the 56 comparisons in this chapter, 52 contain anomalies that can be classified as more anomalous than the training data. And even though we still have some problems with jets from a specific dark matter model (see chapter 8.1), the fact that our algorithms seems to be able to handle every other dataset, suggests that our oneoff networks can help find many new physics models.

## 9.1 Outlook

In this subchapter we want to give a brief overview of every other idea which we had at some point, but were not able to make work in reasonable time, or were just never looked at for some other reason. Additionally to this chapter, appendix F suggests more creative applications of our autoencoder code, including some easy to follow tutorials. Also each dataset in chapter 8 might profit from an extensive hyperparameter optimization.

- We only used simulated data. It might be interesting to see if actual detector data works the same, or if maybe our network misuses some feature only present in the simulation.
- Changing the loss function.
  - You could use actual image like losses (Calculate histograms and compare them). This might cost computation time, but give you more comparable results.
  - Create a permutation invariant loss: Defining your loss function in a permutation invariant way allows to remove the sorting algorithm from the networks. This should simplify the training a lot. Sadly our tries for a permutation invariant loss did not produce as good reproductions as the non permutationinvariant versions.
  - As a simple alternative loss function, choosing an L1 loss results in trivial learning networks. You might solve those problems, by using a power between 1 and 2, for example resulting in a L1.5 loss.
  - Improving image like losses (see chapter 4.5.3).
    - \* Switch the roles of momentas and angles.
    - \* Instead of testing different  $c(x)$  functions, you can find an optimal combination of different  $c(x)$ .
    - \* Use a linear part in  $f(d)$ .
- Datasetup.
  - It seems to be easier to work on transformed 4 momenta (like in 3.2), then on classical 4 vectors, explaining this might help make graph autoencoder converges faster and more reliably.
  - You could still add variables like the mass to the input data, which are not very strongly correlated to the other features, and thus should not confuse the autoencoder.
  - You could use autoencoders to test assumptions about symmetry (like the position of the jet in the detector does not matter): is the classifier on symmetrized data at least as good as the one on non symmetrized data.
- Grapa (our graph autoencoder code).
  - It definitely might profit from a bit more NAN debugging.
  - Also the speed should be able to be optimized quite a lot.
  - Adding something that is called attention mechanisms in the literature, might be a better alternative/improvement to the topK graph learning algorithm. This should reduce the number of NANs if used without learnable graphs, and speed up the training with learnable graphs, while allowing for more creative graph autoencoder applications.

- Improve oneoff networks.
  - Make this work better. This will be a really hard task, but should make them able to classify events truly unsupervised and contamination size independent. (The link is quite long, so you need to click on "this" in the digital version of this thesis, or just write me an email at Simon.Kluettermann@gmx.de).
  - Improve the results of the argumentation in E.4.
  - Train oneoffs only on parts of your latent space.
  - Apply oneoffs to other datasets.
    - \* Use the reconstruction error(add to/replace the latent space).
    - \* (add) physically useful variables, like for example the trivial feature from chapter 5.2.1.
- Instead of evaluating a network only by its AUC score, you could use alternative ones (see chapter 3.1), and try to implement oneoff networks that optimize also them.
- Appendices.
  - Explain why in appendix C.4.2 better compression algorithm work worse.
  - Explain the variance in AUCs you get by using better decompression algorithms (see appendices C.4.3 and D.5).
  - Explain the minkowski metric appearing in appendix A.1.
  - Write the algorithms suggested in appendix C.5.
  - Make the inverse update layers of appendix D.2 more stable.

## 9.2 Acknowledgements

I would like to thank Prof. Dr. Michael Krämer for allowing me to write this thesis, Dr. Alexander Mück for beeing the pretty much perfect supervisor, aswell as Thorben Finke for his help, and especcially in generating the data used in chapter 8.1 and for sharing his computation resources.

I would also like to thank Yuriy Popovich for proofreading this thesis, and my friends and family for supporting me, even though this meant listening to probably way too many pointless thoughts about graphs and anomalies.

Simulations were performed with computing resources granted by RWTH Aachen University under project thes0678.

## Appendices

”Beware all ye who read here” The following chapters might provide more precision on some interesting points, but they often got less attention than the previous ones, and might contain deep dives into topics that do not really affect the content of this thesis. And even though we don’t think that anything in the following is wrong, these chapters may grammatically/visually be less than perfect.

## A Understanding certain choices

### A.1 Changing the input feature space

Referenced in: [3.2] [9.1] [B.3]

The choice of input features described in chapter 3.2 is by no means unique. The only demand at our input data points we have, is that each particle is not compressible without its neighbours. This suggests that simply adding features might not be the best idea, but that we should replace the feature vectors entirely. The alternative way we want to discuss here is just taking the usual 4 momenta.

These data points contain more physically useless information (the position of the jet relative to the detector should not matter) and the distance that is generated in appendix B.3 for the topK algorithm is no longer easily accessible.

We care here about two fairly general results:

First, the metric used in those networks seems to take a peculiar form.

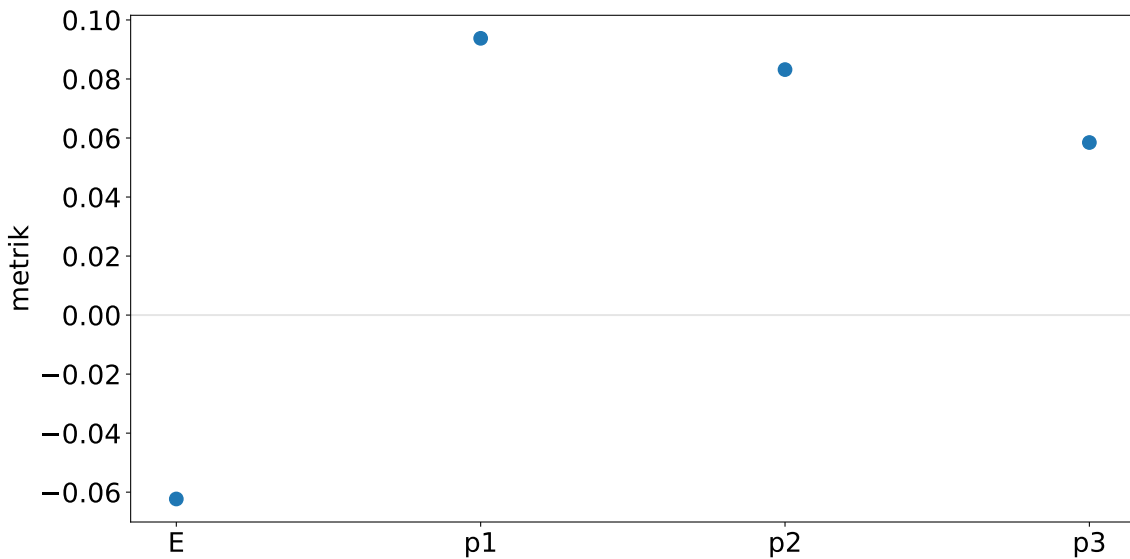


Figure A.1: Metrik of a topK layer for a 4 momenta input

Figure A.1 shows two interesting things: First of, the first value, corresponding to the jet energy, is negative. In the current implementation, this means, that two points are more likely connected, the more different their energy is. You could interpret this as unphysical and try to justify using this that this choice of input parameters is not a good one<sup>99</sup>, but you could also note, that this difference is essentially a minkowski metric. That being said, we are not sure, if this has any significance, as compared to the other monkowski metric that appears in this

<sup>99</sup>You can much more easily justify this, since the resulting training is way less stable.

thesis (appendix E.4.2), this cannot simply be interpreted as a mass, since

$$-(E_1 - E_2)^2 + (p_1 - p_2)^2 \quad (\text{A.1})$$

does not really have a meaning, that is known to us. So maybe this is just random.

As a second point, the loss function in figure A.2 shows a much less useful training curve.

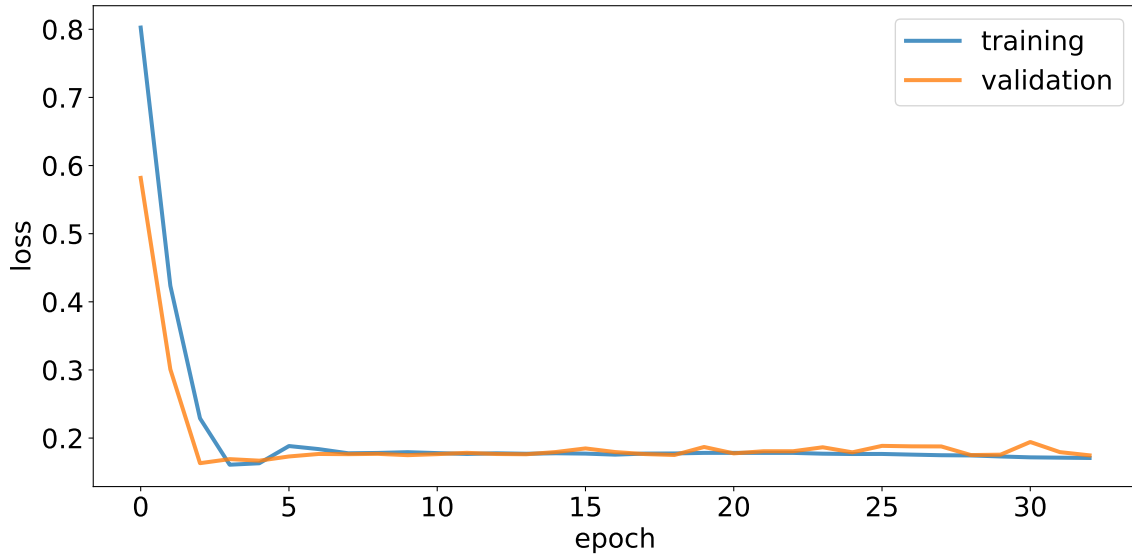


Figure A.2: Training history for the 4 momentum input.

Instead of training for many epochs, it only shows improvement in the first few epochs. This seems to be a consequence of the fact that training such a network seems to be less stable than the usual ones. But if this is just a consequence of our lack of experience with this dataset or if this is a general feature is anyone's guess. Finally, this at least suggests that the input choice introduced in chapter 3.2 is not a bad idea.

## A.2 Is it a good idea to relearn the graph at each step?

Referenced in: [A.6]

Quick answer: No. Long answer: Probably not, not because the quality is necessarily worse, but because the number of NaNs (appendix B.2.2) increases a lot, making training for an effective time very hard and thus resulting in worse classifiers. That being said, this still means, that if you could handle the NaNs, you might profit from more gtopk layer, but we are not able to test this at the moment, and even though multiple different graphs would allow you to see them as something similar to activations, we don't know of any physically useful definition of similarity in angles and momenta, except for the angles themselves (and maybe momenta to a lower degree), so changing the graph setup in the middle of the layers, might not have any effect at all.

### A.3 The consequences of sorting outputs by $l_{p_T}$

Referenced in: [4.4.1] [C.2.2] [C.5] [D.3]

Sorting nodes at the end of the autoencoder breaks the permutation symmetry that we praised at the beginning of this thesis (chapter 2.3), and since we use  $l_{p_T}$  for this sorting, we artificially inflate the importance of the momentum variable compared to the other variables. That being said, turning of this sorting, does hurt the network performance: Given two networks as comparison for this, the sorted network reduces an AUC value (trained on QCD) of 0.6351 into 0.5788 and probably even more importantly, the training curve looks way worse: Compare figures A.3 and A.4.

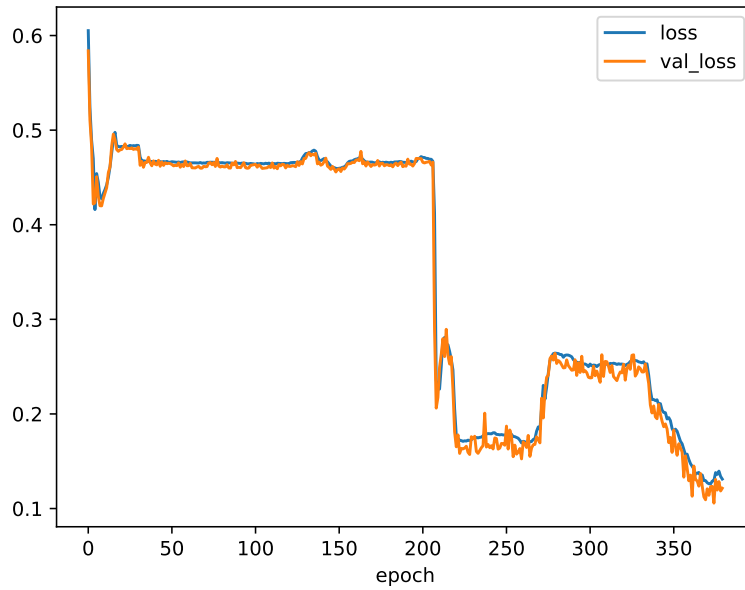


Figure A.3: Training curve using sorting

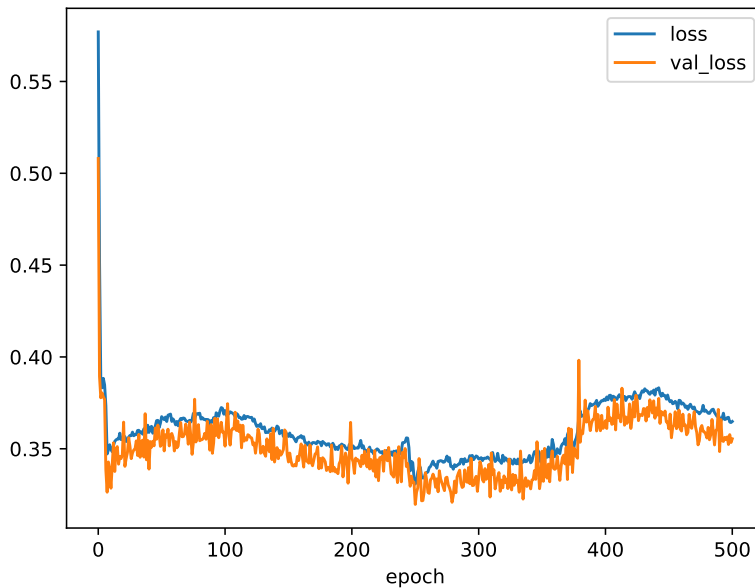


Figure A.4: Training curve using without sorting

You might consider the not sorted curve more clean, but it also does not really improve any further at a fairly early epoch, result in the sorting network reaching a loss of about a factor 3 smaller. Seeing this, sorting seems like a clear choice for us, and so basically all networks in this thesis are sorted. That being said, the original deficits of breaking permutation symmetry, could actually be interpreted to mean the opposite: while it is true, that switching each value, except the sorted one, would not result in the same loss, switching any whole node position, still would result in exactly the same loss. In fact, we can use this, to understand why not sorted networks are so bad: The encoding includes a random<sup>100</sup> node permutation, while the decoding does not, so after the autoencoder, the result is a random permutation of the input features, which then are compared to the still initially ordered input features. That this does not work that well should be clear: So either choose the momentum axis as sorting value<sup>101</sup> or compare your predictions nearly randomly. In fact, you could argue, that this breaks permutation symmetry, as you impose a defined ordering on your node indices. Finally, if you would want to improve this, you might look at two things: making the comparison variable learnable, would remove the artificial inflated importance of  $lp_T$ <sup>102</sup> and making the decompression change the node ordering, in the best case in a learnable way, would make this whole discussion moot, as the network could converge as good with, as without sorting. That being suggested, implementing this is not necessarily easy, as you would not want any function to apply to all nodes, to make sure you don't break permutation symmetry, which for me looks like you restrict yourself to finding a variable to sort by and to reverse an initial sorting would in general not be easy at all, as the initial sorting could be completely random, but would result here probably in a network sorting the nodes by their transverse momentum, as this is the sorting of the initial data, but this seems to us, as a more complicated implementation of our final sorting layer<sup>103104</sup>. So finally: sorting seems to be the right choice for us, but a more advanced algorithm, might still be useful: consider the data from appendix F.3: sorting by one parameter is not that useful, when you only have boolean datapoints<sup>105</sup>

## A.4 The usage of a batchNormalization layer in the middle of the graph autoencoder

Referenced in: [4.4.1]

BatchNormalization layers usually are used to speed up the convergence of your network, but result in usually more chaotic training, so here a brief comparison of a network with and

<sup>100</sup>Actually not random, but you could see it like this, if you only see the initial and final node indices.

<sup>101</sup>Which would not be what you would want, since locality in real space is much more important than similar energies, as appendix B.3 shows.

<sup>102</sup>But maybe also make the training less stable, and add a less controllable importance to some other mixture of features.

<sup>103</sup>That could actually work less well, not only since it needs more calculation time, but also since this sorting is done at deconstruction, meaning that later graph update layer won't have an effect on it.

<sup>104</sup>You might also ask yourself if you could not just remove the permutation from the encoding layer, but this is easier said than done: As it is true, that the sorting is generally just done for implementation, but as you combine 4 values into for example two, you could have situations, in which node 0 and 3, as well as node 1 and 2 are combined together into (0,3) and (1,2), and even without sorting, reconstruction this, would result in 0,3,1,2, so you would still either need some kind of permutation in the decompression, or some kind of shortcuts between the layers, that encodes the original position: This would not be bad style, as it could result in the network learning to misuse this information to encode arbitrary information, but would also not be very easy to implement, and might require a nonpermutation invariant compression and decompression function to work well, which would obviously not be ideal, as keeping permutation invariance is the main reason for this chapter.

<sup>105</sup>Even though in this chapter no real sorting was used, and you could still work with our approach and multiple sorting layers fairly well, assuming `tf.math.top_k` is stable (their documentation does not say so, but the implementation is, but this may change since also `tf.argsort` is stable at the time of writing this, but they want to implement a not stable version later to improve the speed of this algorithm).

without a BatchNormalisation

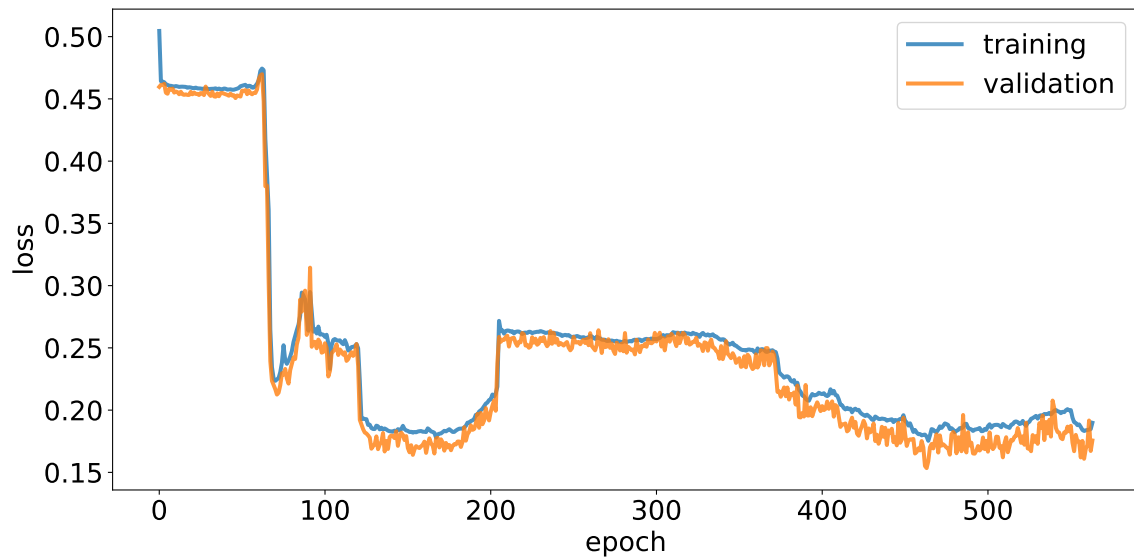


Figure A.5: Training history with a batchnormalization layer

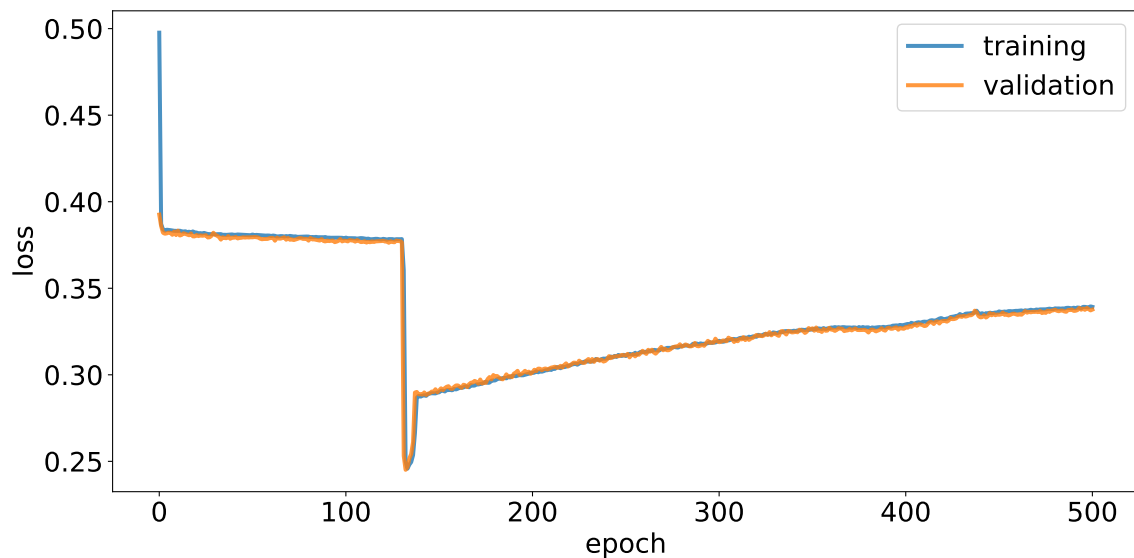


Figure A.6: Training history without a batchnormalization layer

If you compare figures A.5 and A.6 you see, that the loss is clearly better with a batchnormalization layer

So even though you might not be able to induce this from a single test only, Batchnormalizations seem like a good idea. As a sidenote, using a batchNormalization before the comparison is usually not a good idea, since this normalization includes a learnable scale, which results in the network only learning to compare zeros to zeros.

## A.5 Changing the definition of the transverse momentum input

Referenced in: [3.2]

If you are familiar with image based neuronal networks, choosing our momentum preprocessing might seem a bit strange. Since  $-\log(x)$  is monotonously falling, low momenta correspond to high  $lp_T$  values. And since image based networks weight each part with the absolute value of this value, this seems like no good idea (to understand this further, see chapter 4.5.3). But graph neuronal networks don't weigh a loss with its transverse momentum, so we don't expect this to be a problem<sup>106</sup>.

To test this, we train a network similar to those from chapter 7.3 with  $\log(p_T + 1)$  instead<sup>107</sup> of  $lp_T$ . This would also include information over the total jet momentum, but since we still use normalization, this mostly gets filtered out automatically. Comparing both networks is not easy, as the loss is defined differently, and the reconstruction is nearly perfect. So we simply use the AUC (with oneoffs): With  $lp_T$  we reach an AUC score of 0.635 and with  $\log(p_T + 1)$  we get an AUC score of 0.621 on the same model. So we get very slightly worse AUC scores and use  $lp_T$  thanks to this. It should be noted that this is not the most effective test, as hyperparameter or just repetition could change this completely, but it does not seem to have a big effect anyway.

## A.6 Comparing our graph update layer to particleNet

Referenced in: [4.1.1]

There are multiple different ways of implementing a graph update layer, a notable one is the one used by ParticleNet [44]: Their graph connectivity is implemented, by just storing all neighbouring vector to each given vector in a set of vectors, this means, they can implement the update procedure as a function of the original and a vector representing its neighborhood<sup>108</sup>. This is not exactly what we do here, mostly since the implementation of the graph as just a corresponding set of neighbourvectors demands for computational reasons that each node is connected to a same number of other nodes, and also requires relearning your graph after each step, which we don't want to force our network to do, as explained in appendix A.2. And also this would make this less of a graph autoencoder, and more into an autoencoder with some graph update layers in front of it, since there is no way to reduce the number of nodes for such an implementation, without completely ignoring the graph structure. Please note the difference: Since we use an adjacency matrix itself to define the graph(in comparison to calculating some derivative from it), you not only have complete control over the graph, that can be used to shrink the graph structure with the number of feature vectors, but you also allow for an arbitrary number of connections for each node<sup>109</sup>.

<sup>106</sup>This story becomes a lot more complicated when using image like losses, but generally, as long as the alternative loss is not related to  $lp_T$  this is not a problem here either.

<sup>107</sup>We add 1 to keep each value positive and to remove divergences.

<sup>108</sup>This function is actually a bit complicated, involving not only convolutions, but also normalisations between them, and they end by concatting the updated vector to the original one, which is something that is not very useful, when you want to reduce the size of your graph.

<sup>109</sup>This is mostly interesting, since it extends the number of possible compression algorithms: They do not anymore have to satisfy keeping the number of connections constant: The number of possible graphs with  $n$  nodes is  $2^{n \cdot (n-1)/2}$  (ignoring permutation invariance, self connectivity and directed graphs), for  $n = 4$  this results in 64 possible graphs, of which only 6 are of this kind. This means that much less compressed graphs are possible, and that finding an algorithm, that can pick only those graphs, is much more complicated (see appendix B.4.2 for more).

## B Experiments using graph autoencoder

### B.1 Varying the compression size

Referenced in: [C.2.2] [D.1]

Compression sizes are usually a bit arbitrary (for our solution to this, see appendix D.1) and so to find an optimal compression size, testing every possible one is generally a good idea. See for this figures B.1 and B.2. We do this here for non normalized networks, so our choice that we extracted from this analysis whas to choose a compression size of 7.

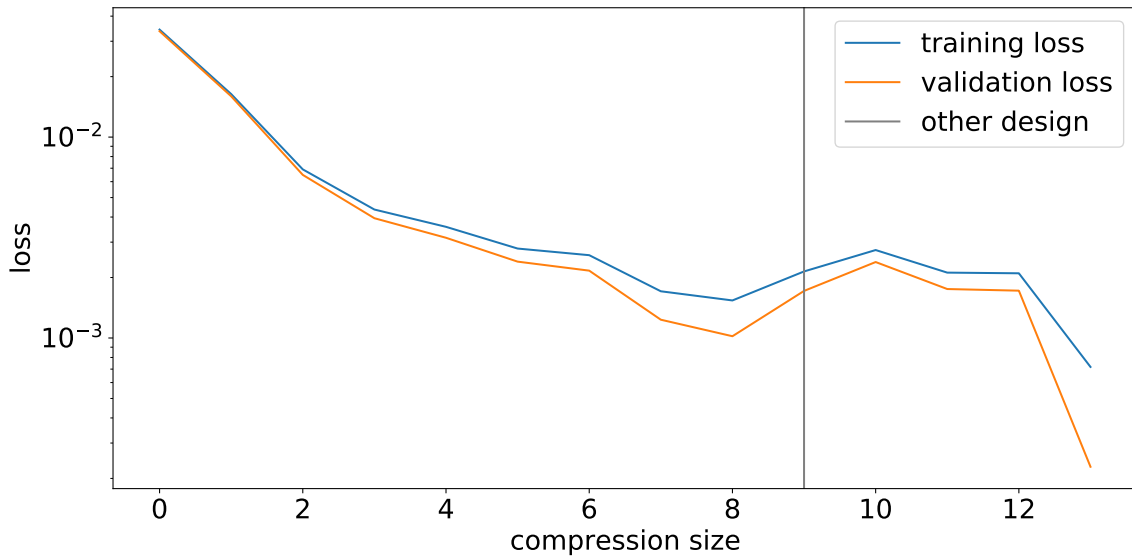


Figure B.1: Rastering each compression size for a 4 node network (normalized) and showing each loss

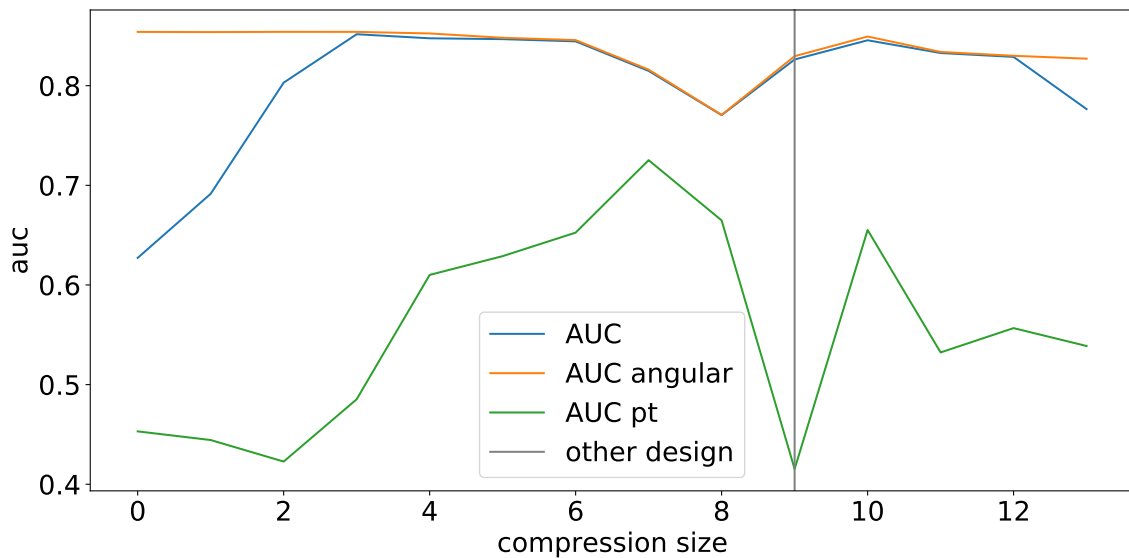


Figure B.2: Rastering each compression size for a 4 node network (normalized) and showing each AUC value (without oneoff networks)

For latent spaces of at least 9, we use a slightly different network setup, adding additional parameters in 2 steps. You see the effects of this slightly in figure B.2.

## B.2 Things we learned from implementing a Graph Autoencoder in tensorflow and keras

### B.2.1 Overflow in angular differences, and how to solve it

Referenced in: [3.2]

Our input data contains a  $\phi$  that is centered around its mean: so the most simple implementation would simply subtract the mean of  $\phi$  from each  $\phi$  value. This can lead to overflow problems, since  $\phi = 2 \cdot \pi$  is equivalent to  $\phi = 0$  and thus a mean of about 6 could be subtracted from a tiny value<sup>110</sup> resulting in weird phi distributions. Not solving this, results in a loss distribution with a small peak at very high losses

So how to solve this? first we need the true mean value ( $-0.1 + 2 \cdot \pi$  and  $0.1$  have mean  $0$  and not  $\pi$ ), and then we use a modular operator to restrict every difference (the difference  $-0.1 + 2 \cdot \pi$  modulo  $2 \cdot \pi$  equals the true difference of  $-0.1$ ). And for finding this mean value we can cheat a little, but calculating the mean 4 vector, and finding out its  $\phi$  value. For a more indepth look at our solution, you can also take a look at the actual implementation <https://github.com/psorus/grapa/blob/master/grapa/layers.py#L6488>.

### B.2.2 How to deal with NaNs

Referenced in: [4.5.3] [6.1.2] [A.2]

When setting up your own graph autoencoder, one thing you might have to deal with are networks that return NaNs. We observed three kinds: Networks, which loss goes NaN at some point: These NaNs you can deal with the best, since there is a keras callback `terminateOnNaN` which stops the training from escalating further, and so such NaNs just stop the training a bit too early and make the network just a bit worse. That being said, sometimes these NaNs can be simply avoided by

- Making sure your functions are continuously differentiable, we had a problem with  $\sqrt{|x|}$ , even though it is only not differentiable at a single point, that you probably never reach, but  $\sqrt{|x| + 1} - 1$  works.
- Often enough it can be enough to simply retrain your networks, as some NaNs are fairly rarely called, but if the loss is NaN it usually stays NaN.
- It seems like it helps to reduce the complexity of your function, we saw that while thinking about normalization:  $x / \text{std}(x)$  NaNs, while  $\frac{x}{\max(|x|)}$  does not.

Giving those NaNs, there are two other cases: NaNs in a learnable parameter, that either results in a loss that is NaN or not. Both cases are fairly rare, but appeared. You might ask how a parameter that is NaN can result in a loss that is not NaN, as every function of NaN is NaN, but there are parameters, like the metrik in the topk algorithm, which only affect the ordering, and apparently is the sorting able to handle NaNs<sup>111</sup>. We mention these here, since hidden NaNs are very hard to detect and can still hurt the performance<sup>112</sup> The only way to check for NaNs is to go over each parameter and check it, which we do after each training, but even when doing this, we are not be sure that we did not miss a NaN parameter in an automatically repeated network.

<sup>110</sup>Or the inverse.

<sup>111</sup>Even though we have no idea what the sorting algorithm of tensorflow exactly does to NaNs.

<sup>112</sup>Consider topK NaNs: instead of a useful graph, this graph is just random.

### B.2.3 Why relus are great

Referenced in: [7.1.1]

Activations like  $\text{relu}(x) = x + |x|$  allow a neuronal network to learn nonlinear functions. You can also use them, and relus specifically (since it is probably the easiest one) to implement functions yourself. Consider the following function heaviside function:  $f(x) = 1$  for every  $0 \leq x$  and  $f(x) = 0$  else. Using this function in tensorflow is not easy, notably since it is not differentiable. But you can approximate it quite well as  $-\text{relu}(C \cdot x) + \text{relu}(C \cdot x + 1)$  (using a high value for  $C$ ). This trick is used multiple times in grapa, but you should be careful: Since the higher  $C$  is, the less differentiable this function is, choosing a too high  $C$  might be a bad idea. Also, this specific example can also be implemented as  $\text{sigmoid}(C \cdot x)$ .

## B.3 Metrik analysis

Referenced in: [A.1] [A.3] [B.4.2]

As explained in appendix B.4, our topK algorithm, on which all graphs are based, uses a learnable diagonal metric, which is used to define similarities in the network. This metric can be extracted to understand this sense of similarity. Figure B.3 shows that unnormalized networks use the angular differences between nodes to define similarity. Interestingly figure B.4 suggests that using a normalization changes this. Now networks only use one angle, and a negative metric value for the other one. This means that two nodes are more similar the more one angle is different, but also the more different the other is.

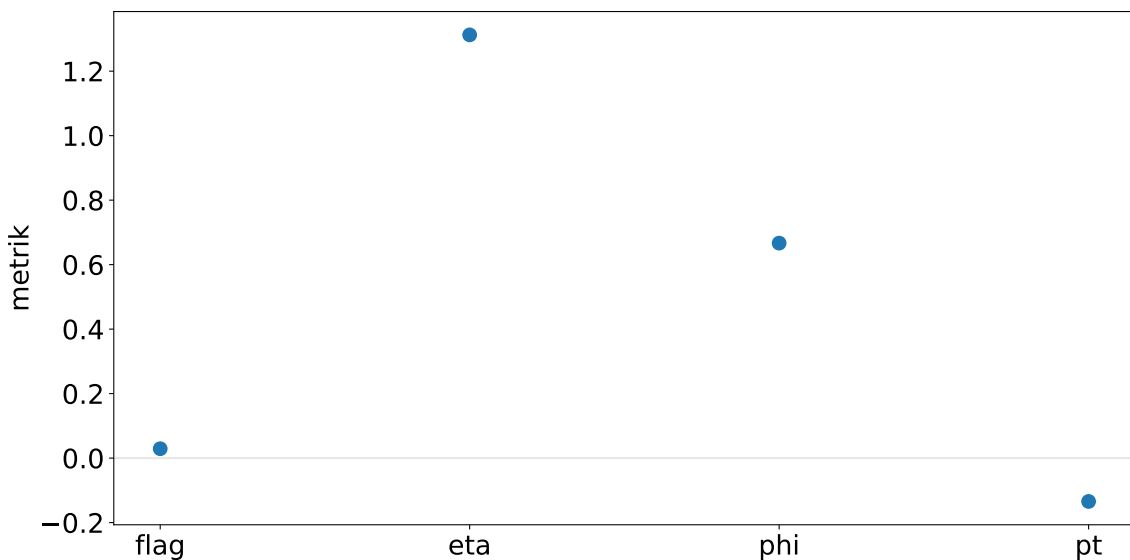


Figure B.3: Typical metrik of unnormalized networks

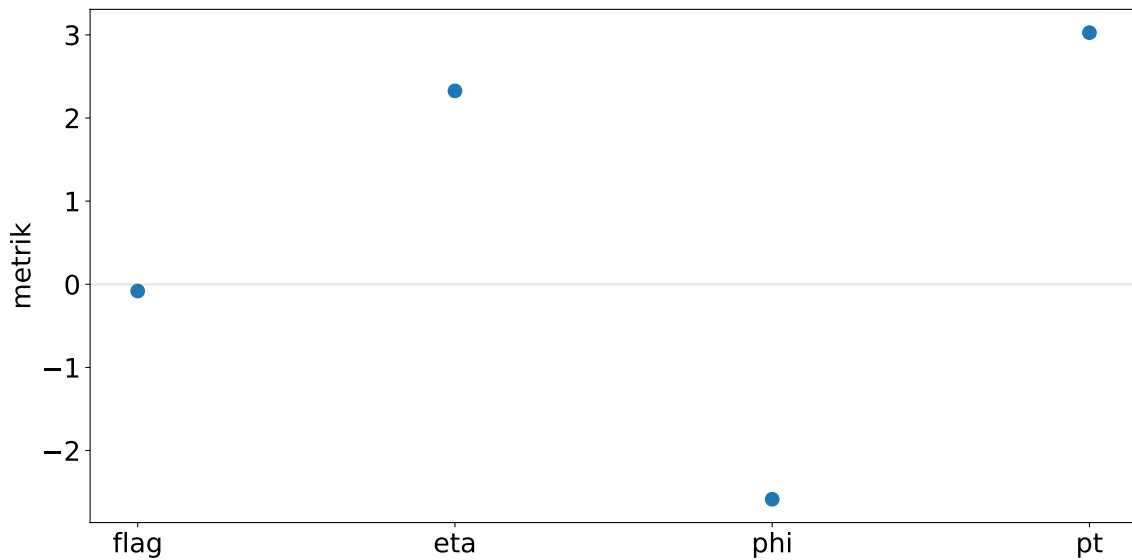


Figure B.4: Typical metrik of normalized networks

For a metrik on different features see appendix A.1.

## B.4 How topK works exactly

Referenced in: [2.3] [4.4.1] [B.3]

The probably most commonly used algorithm, to construct a set of graph connections from a list of vectors, topK, seems to be quite easy to understand: you connect each vector, to the  $K$  vectors that are most similar to it. The difficulty lies in the word similar: Here two vectors are more similar, the smaller the l2 difference is. In an attempt, to make this more powerful, we also use a learnable metric in this l2 difference. Even though this might not be strictly necessary, since the network can change parameters to accommodate its sense of similarity, this still allows the network to better choose what to focus on in each topK layer. It can be quite useful for autoencoder, since for example ignoring a parameter, could else only be done, by decreasing its size in relation to the other parameters, which might not be optimal, when you want an accurate reproduction. This also allows you to create a graph, before having any learnable layers. On the other hand, these metric can complicate the calculation of the adjacency matrix, which we try to manage by demanding that the metric is entirely diagonal, reducing also the needed time drastically, and the parameters of the metric can increase the occurrence of divergences in training, since even a small change of those parameters can affect the network output in huge ways. That being said, having a humanly understandable metric, can lead to interesting insights (see appendix B.3). You could ask yourself, if a topK algorithm is the best choice, since the number of possible adjacency matrices is quite low, see for this appendix B.4.2. Finally, it should be noted, that the topK layer can increase the size of each of the feature vectors, which is useful for the compression algorithm, even though in this specific example this is not used.

### B.4.1 Problems

Writing a topK layer to connect each node just to its  $K$  neighbours is actually not really possible: The idea of a graph in which each node afterwards has  $K$  nodes does not always work, consider the following graph and  $K = 1$  in figure B.5<sup>113</sup>.

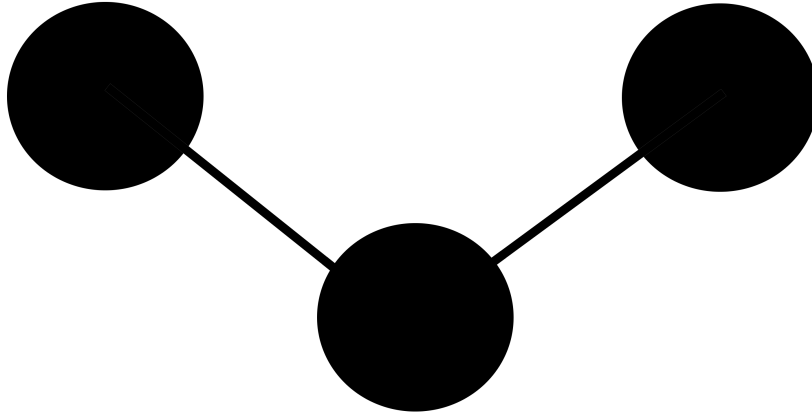


Figure B.5: Example of a set of nodes that cannot perfectly be connected using a topK algorithm (here  $k = 1$ ). The problem here is, that 2 nodes have the same distance. We connect in this case both

When there are two nodes of the same distance, which to connect? We simply connect both, as states in which both are of the exactly same distance are very rare, but there is a more complicated problem in figure B.6.

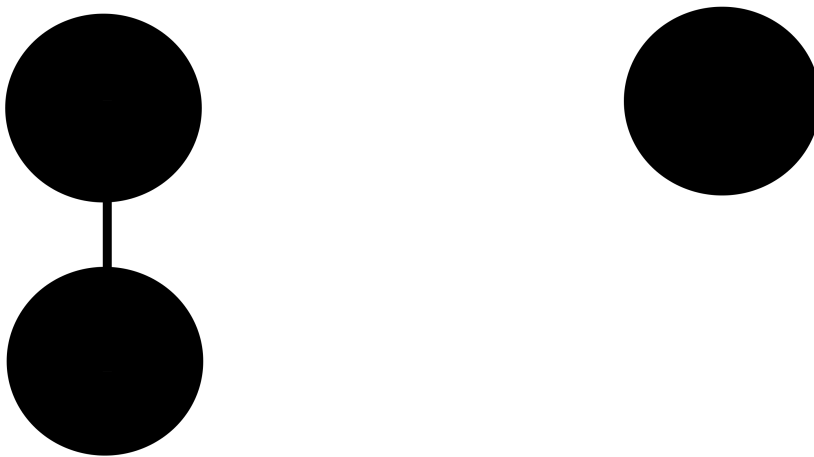


Figure B.6: Example of a set of nodes that cannot perfectly be connected using a topK algorithm (here  $k = 1$ ). The problem here is, we want to connect the last node to a node that has no more open connections. We solve this by using asymmetric adjacency matrices

<sup>113</sup>Choosing a low  $K$  is done here to make the example easily accessible.

How to connect to nodes that don't have neighbors open? We solve this by no longer requiring the adjacency matrix to be symmetric, and thus allowing the graph to be directed.

### B.4.2 Why topK might actually not be the best idea

Referenced in: [A.6] [B.4.2] [F.2.3]

Since this is just an appendix, I want to take the time, of starting this chapter with a story: A while ago, I was in a chapel. This does not happen very often, and judging from the chapel, it is also not used very often. One thing you immediately notice, is that this chapel did not utilize church banks, but just used a lot of chairs, and since at that moment, I desperately wanted to think about something else, I became fascinated by those chairs: For some reason, even though these chairs are fairly unordered, I didn't think of them as just  $n$  chairs, but as  $r$  rows of  $c$  chairs each. Why? How do we abstract those unordered amount of chairs into need rows and columns, and do we have an algorithm that can do this for us? Since this is still a thesis about graph autoencoder you might guess the relation: if you simply draw graph connection between each neighbouring chair, you generate a graph like in figure B.7.

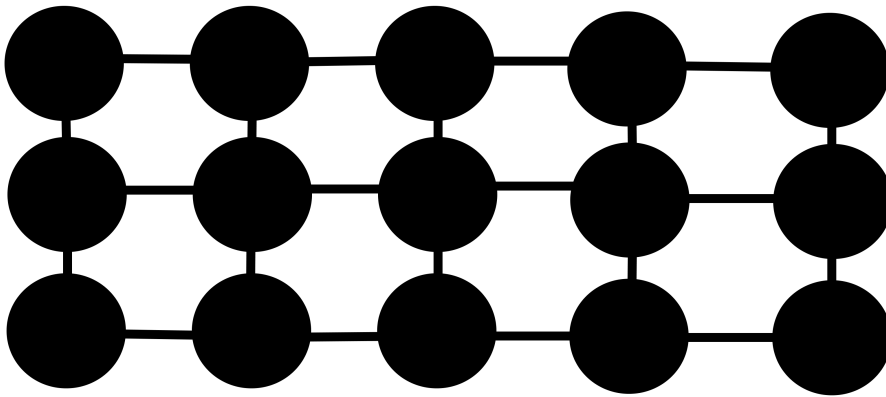


Figure B.7: A graph representing chairs.

This graph is easily abstracted and reconstructed using the algorithms explained in chapters 4.2 and 4.3: You can use one axis of the diagram (probably  $y$ ) as separation variable. This results in subgraphs for each row, which get compressed into one object that we might call a row. Then, this row graph tensorproducted to the sitting plan as graph of rows gives the original graph. There are problems, notably you need to know the number of rows before<sup>114</sup>, but this is still an application of a point cloud, that is well compressed using my graph autoencoder. There is just one problem: the graph we use is not a topK graph! To be precise: A topK graph is defined by each node having the same number of neighbors, but the chairs next to the edge have fewer neighbors than chairs in the middle of the chapel. This means that this graph cannot be the output of the topK algorithm. And you can show quite easily, that the tensor product of a topK graph with a topK graph is the only way to return a topK graph.

So how to solve this? One way would be to connect everything below a fixed distance, but testing this requires more effective high node networks, as in our tests this did not make any difference. Also choosing a fixed distance is an arbitrary choice which we don't like.

<sup>114</sup>As the compression number has to be a factor of the number of chairs, and finding half rows or double rows, still would be good representation, this is actually not that big of a problem.

## B.5 Trainingsize, and why graph autoencoder don't care about it

Referenced in: [4.4.1] [4.7.2] [7.4.2] [C.2.2] [D.4]

Graph networks with oneoff networks show some properties that a usual network don't. One thing is the apparent independence of the training size, and even when you can easily explain this, as having few parameters in your network, this still might allow you to train on data that was not usable before (see chapter F). We show training histories in figures B.8 and B.9, while figures B.10 and B.11 show that this requires oneoff networks.

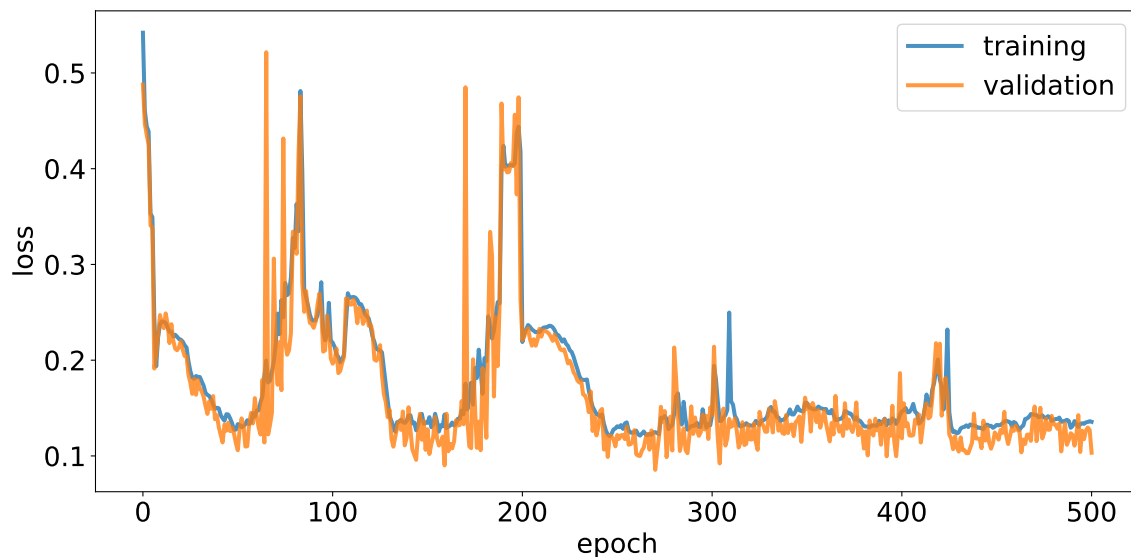


Figure B.8: Training curve trained on 50k top jets

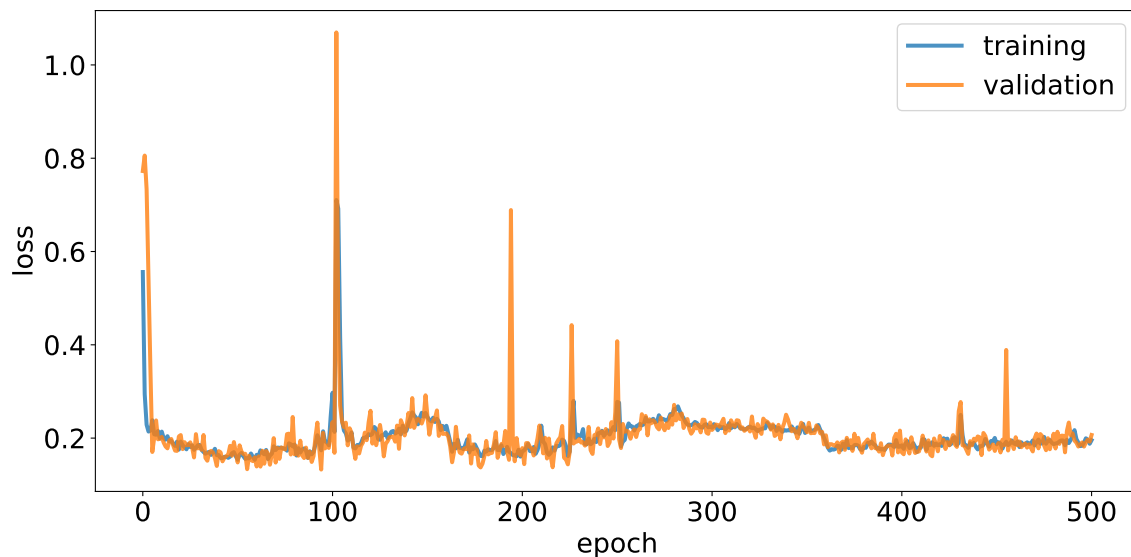


Figure B.9: Training curve trained on 5k top jets

The training history for 5000 jets seems actually even less noisy. This is why we can use a reduced size to remove NaNs (see chapter 7.4.2).

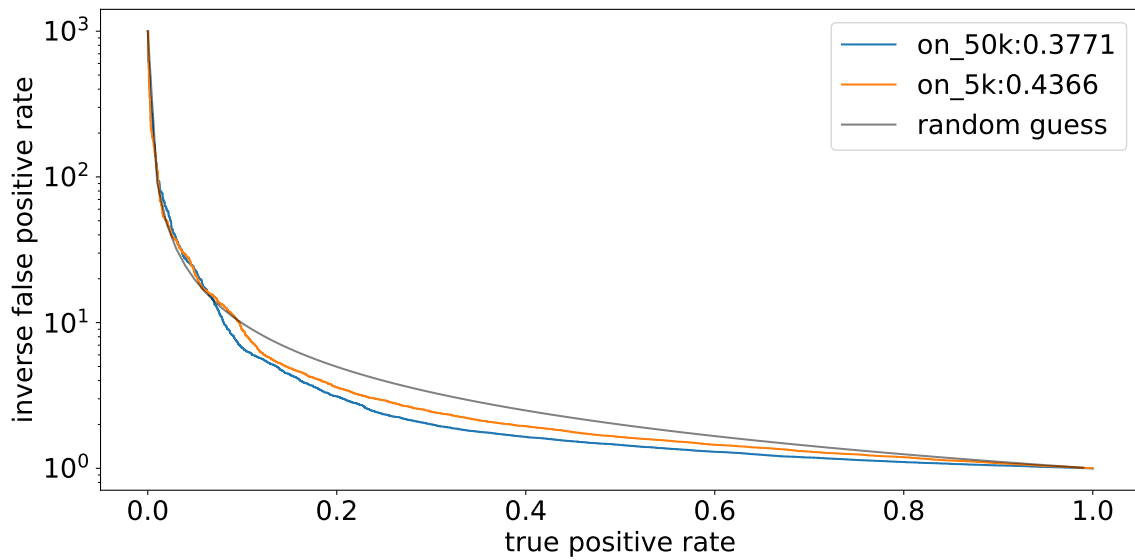


Figure B.10: Double ROC curve for different training sizes

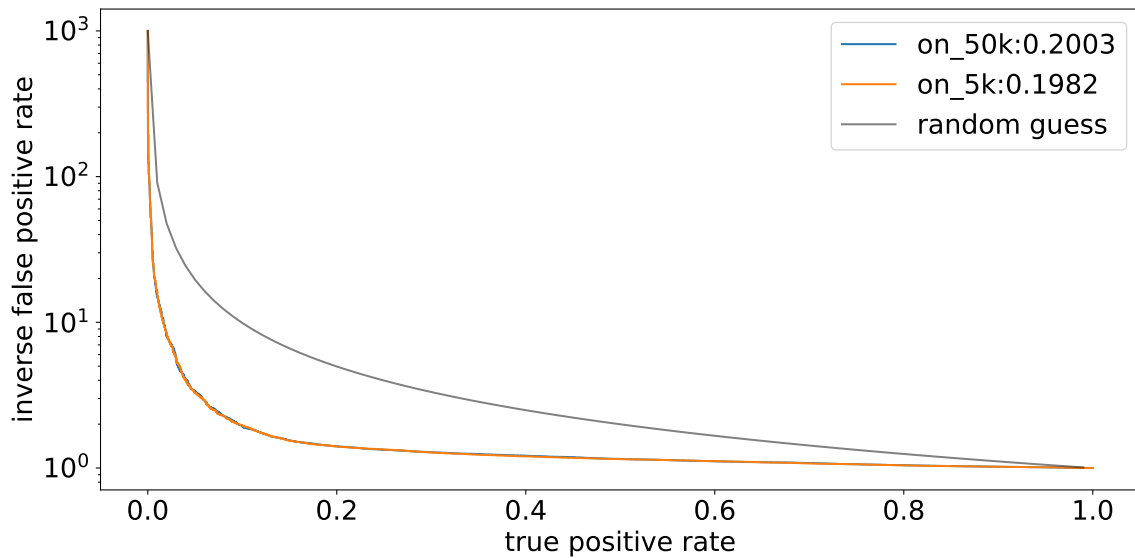


Figure B.11: Double ROC curve on oneoff networks comparing training sizes. You should see no actual difference here

## B.6 Why autoencoder reproduce mean values

Referenced in: [6.1.2]

One thing you often see your autoencoder do, is learn mean values of distributions instead of the whole distribution. This you see most clearly as the width of each output distribution being smaller than its input distribution, and even though these distributions are not exactly constant this is still the same effect, which is why we want to explain it here a bit more closely. As mentioned in chapter 4.5 you most prominently see this effect with a l2 loss, and you can easily see why: consider an easy example of a network either requiring  $-1$  or  $1$  and being able to output every value in between. Let's say it guesses either  $1$  or  $-1$ . With an accuracy of  $a$ , then the loss is  $4 \cdot (1 - q)$  if on the other hand, it guesses  $b \cdot \text{value}$  where  $\text{value}$  is its rounded guess, then the loss becomes  $q \cdot (1 - b)^2 + (1 - q) \cdot (b + 1)^2$  which can be smaller than the original loss for every  $q \leq 1$ , being minimal at  $b = 2 \cdot q - 1$ , which for every  $q \leq 1$  is not equal to  $1$ , and

thus choosing a prediction smaller than the true prediction is beneficial in minimizing the loss. Also for  $q = 1/2$  you see the prediction to regress to 0 and thus the network to learn the mean of the network. This also explains why at the beginning of the training, where the network does not really have a clue what the real distribution is, the first thing learned is the mean of this distribution.

## C Overview of less useful networks

### C.1 Failed approaches

Referenced in: [2.4]

In this chapter we will quickly go over some bad ideas you could have, on how to implement a graph autoencoder and finish with the first implementation that could be considered working at least a bit. These implementations are usually defined by an encoding and a decoding algorithm, so basically something to go from a big graph to a small graph, and something to reverse this again. In addition to this, the graph update and the graph construction stay mostly the same as it was explained in chapter 4.1 and 4.4.

#### C.1.1 Trivial models

Let us start with the probably most simple autoencoder algorithms: To make a  $n$  node graph into a  $m$  node graph, we just cut away the last nodes until there are only  $m$  nodes left<sup>115</sup> to reduce the graph size, and add zero valued particles to it again. One difficulty here lies in the fact that those particles have no more graph connections, this we solved by just keeping the original graph connections stored. Sadly, those networks just don't work: even when we would set the compression size over the input size, the reproduced jets hardly bare any resemble to the input jets: This is the first example of the central problem of graph autoencoding: Permutation invariance. Consider the following encoder: two numbers  $a$  and  $b$  where  $a = b + 1$ , this would be trivial to compress into one number for a normal(dense) Autoencoder(maybe just take  $a$ ), but here we have to respect permutation symmetry, so basically we do not know what the first and what the second particle is and how do we decompress now? In this context you could keep one of the parameters and try to encode if the other one is bigger or smaller than this, maybe you also know that  $0 \leq a$  and you could multiply it by  $-1$  if it is the smaller one, but this is less than trivial, and by increasing the number of parameters this gets even more complicated. This is a problem that mostly appears as the inability of even a "good" Autoencoder to work with and compression size that is equal to the input size, building an identity (see appendix D.2). That beeing said, permutation invariance can also be a benefit, especially in permutation invariant input data, more to this in appendix D.3

#### C.1.2 Minimal models

To improve this model, we started working with smaller graph sizes (mostly the first 4 particles), making the structure less complicated, and allowing for more experimentation thanks to the lower time cost. Notable improvements include replacing the added zeros by a learnable function of the remaining parameters, relearning the graph on the new parameter space and adding some dense layers after the graph interactions, but the most important improvement was achieved by making the compression and decompression local in some learning axis. Instead of just removing parameters in an arbitrary way of physical intuition, we demand that particles which are similar in some way are to be compressed together: This is achieved by the creation of a function that compresses a set of particles into one particle, and allow the network to learn what similarity means<sup>116</sup>. These networks still have problems, as we will discuss in the following,

<sup>115</sup>Please note the importance of the  $p_T$  ordering here: Cutting the last particles means cutting the particles with lowest  $p_T$  and thus the probably least important particles.

<sup>116</sup>In the compression step, we define a new feature for each node, by which we sort the set of nodes, and afterwards we build sets of  $n$  particles from this ordering, and compress them using a linear function (it might be interesting to look at nonlinear functions, but we generally see worse results by adding an alinearity). Please note that since we use a feature to sort the elements, and in the graph update step there are neighbour steps, that generally increase similarity, connected particles are more probably compressed together, even though we

but generally produce respectable decision qualities, and show similarities between input and output image. These network is discussed in the next subchapter.

## C.2 The first graph autoencoder that could be considered working

Referenced in: [3.3.2]

This autoencoder takes QCD jets, transforms them as introduced in 3.2, does some additional preprocessing, after which a graph is constructed, a graph update step is run, after which a graph compression algorithm is applied just to be reversed afterwards, reconstructing a new graph, which is again used to update the feature vector, after which (and after a sorting step), the current graph is compared.

After this graph compression, which reduces 4 times  $flag + 3$  parameters into one vector with 10 parameters, we use 3 dense layers reducing the parameter count down to  $6^{117}$

### C.2.1 Training setup

Another thing that has to be clarified concerning this model, is the training procedure. We use the adam optimizer, with a learning rate of 0.001, with a batch size of 200 and train the network, using an earlyStopping callback, until it does no longer improve its validation loss for 10 epochs and afterwards use the epoch with the minimal validation loss. We use 600000 training and 200000 validation jets to plot here the loss for each epoch in figure C.1.

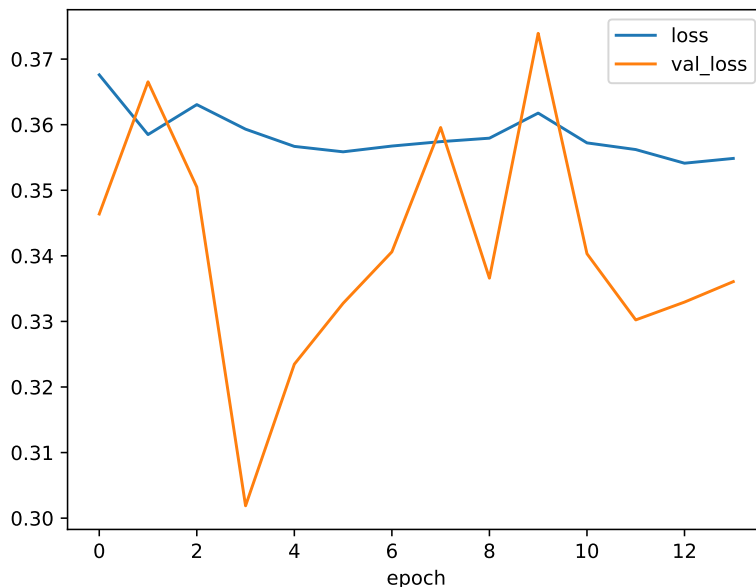


Figure C.1: Training history for first working autoencoder

As you see, there is not really any progress made in the training<sup>118</sup>, but you already see one fact, that will be quite common in the following: The validation loss is not (much) bigger than the training loss, neither at the end, not anywhere. This is fairly uncommon, as usually earlyStopping is used to combat overfitting, and validation losses that seem to increase at some point, but also easily explained, since encoder and decoder only amount to a total of 840

do not demand this.

<sup>117</sup>Contrary to usual autoencoder, choosing a compression size as small as possible does not really matter to us, since the classification power has a complicated relation to it (see appendix B.1).

<sup>118</sup>Except for maybe the first epoch, which is not shown in these kind of plots.

trainable parameters, which is not enough to store information for  $O(1)$  events. Interestingly, this seems to be a clear benefit for graph autoencoder, as even bigger networks with similar amounts of parameters, trained on fewer data, don't seem to show any tendency to overfit. This allows us to reduce the training size to at least 2 orders of magnitude less, without any quality loss (see appendix B.5), and you could even ask yourself if it would not be possible to remove the whole need of splitting your data into training and validation data. That being said, this dataseparation is maintained for the rest of the thesis, and this overfitting safety comes at a price: the validation loss might not increase in relation to the training loss, but that does not mean that both cannot increase in parallel. This, and the fact that graph training curves are way more noisy than usual training curves, make earlyStopping still a viable training callback, and result in most of the reasons, each training stops.

### C.2.2 Results

So why do we consider this the first working model? This might be the first model, that can show some resemblance between the input and the output in figure C.2.

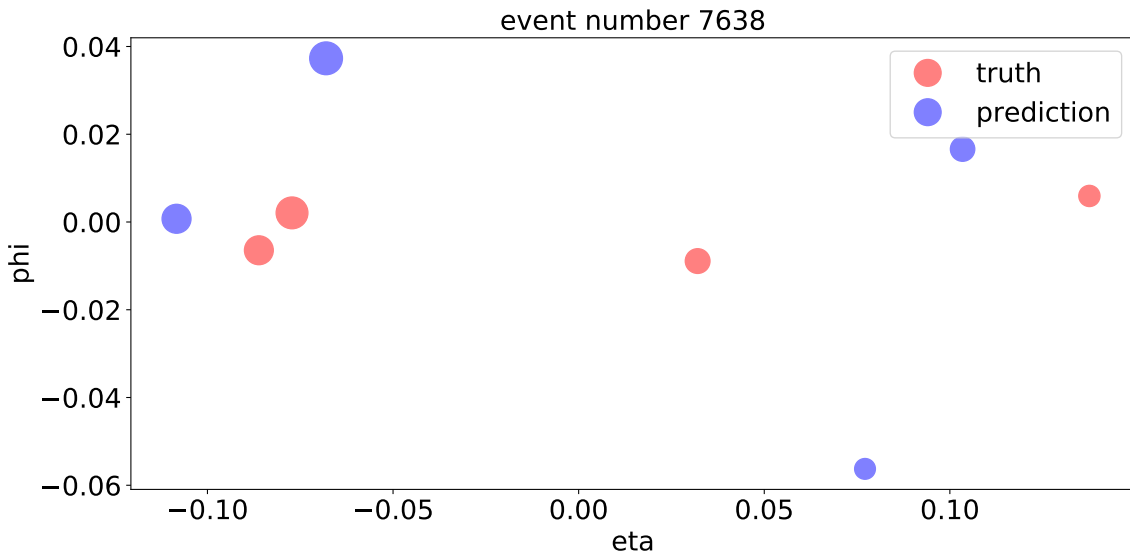


Figure C.2: A reconstruction image for this model, we choose here one of the best reconstructed events

And even though this resemblance is not very good, and this is even one of the better examples. This might also be the first model, that consistently returns finite values, which was not necessarily trivial at the beginning. But most importantly, this network already returns a seemingly good AUC score of over 0.8, which for taking only 4 particles seems to be quite impressive<sup>119</sup>

<sup>119</sup>A bit too impressive actually, see chapter 5.

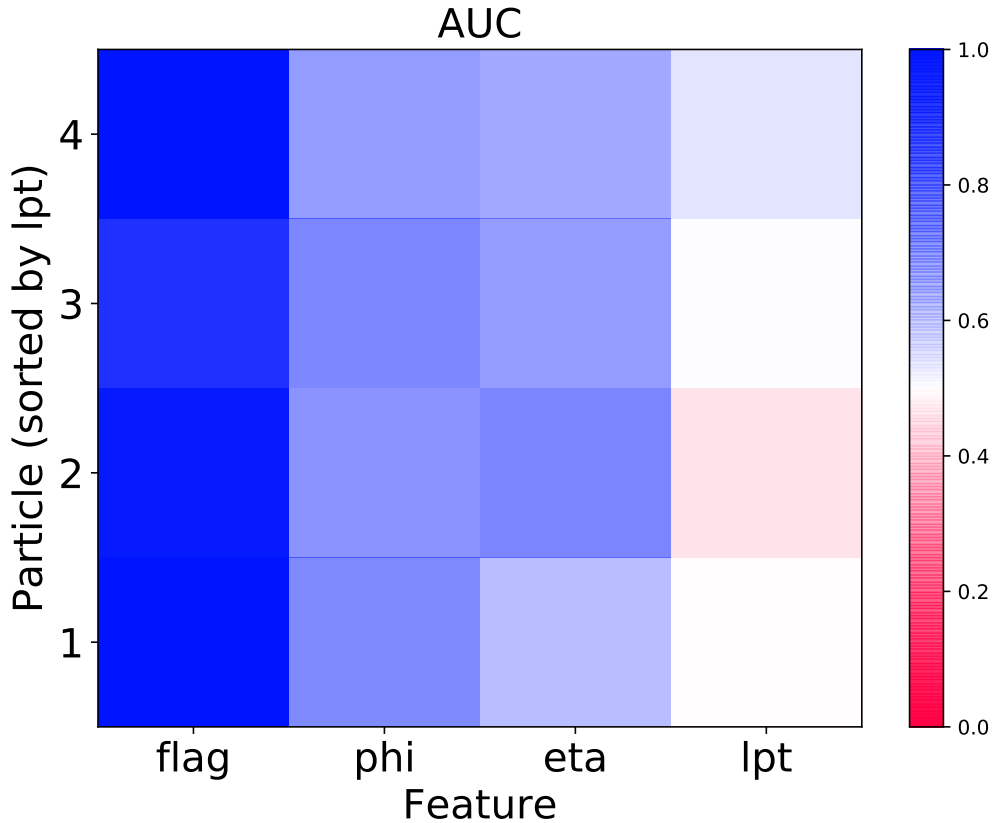


Figure C.3: AUC feature map for this model

If we look at the featuremap in figure C.3, the first thing you should notice is the perfect score in the flag column. This is sadly just numerical noise, that happens, when a reconstruction is nearly perfect, which sometimes happen, when the autoencoder just copies a value, and not very useful if you want to differentiate between jets. So ignoring this, the angular columns seem useful for comparison, and as if most of the classification power comes from here. Finally, the momentum column seems fairly useless as it has AUC's below and above 0.5.

### C.3 Improving autoencoder

Referenced in: [3.3.2]

Given the fairly good AUC score, it looks like the only thing we now need to do, is to increase the size of this autoencoder, and we probably have a really great anomaly detection algorithm. But before we try, and fail, at this, let us improve our autoencoder first. As you might agree, the training curve does not look very impressive, and the reconstruction is also not very good. Thats why we suggest some changed model<sup>120</sup>.

#### C.3.1 Training setup

Here we still train on the first 4 nodes, and with the same batch size of 200, but with a lower learning rate of 0.0003 and with a higher patience, stopping only after the network does not improve for 30 epochs. We also increase the compression size from 6 to 7.

<sup>120</sup>We alter models iteratively, but since we don't want to show tausends of models here, you only see summaries, which is why the changes seem a bit random.

### C.3.2 Results

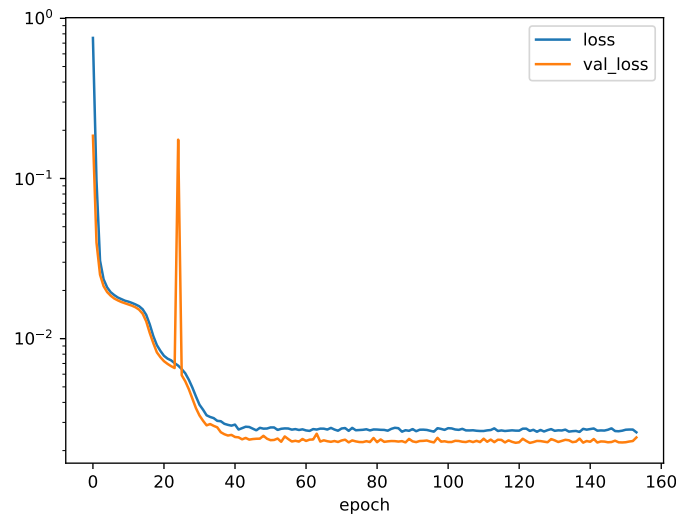


Figure C.4: Training history for a better autoencoder

As you see in figure C.4, the training history looks way better: The training does not stop after only a few epochs<sup>121</sup>, and it is less random than before, while still not showing any signs of overfitting. On the other hand, you might notice a peak in the validation loss at epoch 24. Sadly this is fairly common: The validation loss is not worse, since many events are slightly less good reconstructed, but through very few really inaccurate ones. This might be a consequence of some slight variation of a parameter, that completely changes the reconstruction in some edge case. The metric of the topK algorithm or the sorting feature for the compression might be such parameters. Luckily those peaks disappear, when the problematic parameter has a less dangerous value, and since we always take the model with the minimal validation loss, we can just ignore those peaks<sup>122</sup>. You also might notice that the learning is done basically completely in steps: the loss is constant for some time, after which it suddenly improves rapidly. Finally, please notice, that the most improvement is done in the very first epochs. This is quite common, up to networks that do 0.9 of their training before the first validation loss is calculated.

Also the roc curve improves, reaching an AUC score of over 0.85

<sup>121</sup>And this is not only an effect of the patience being bigger.

<sup>122</sup>You can control the number of peaks by tweaking the learning rate, but a lower learning rate generally keeps the loss from improving further at some point, and thus might not be a good idea.

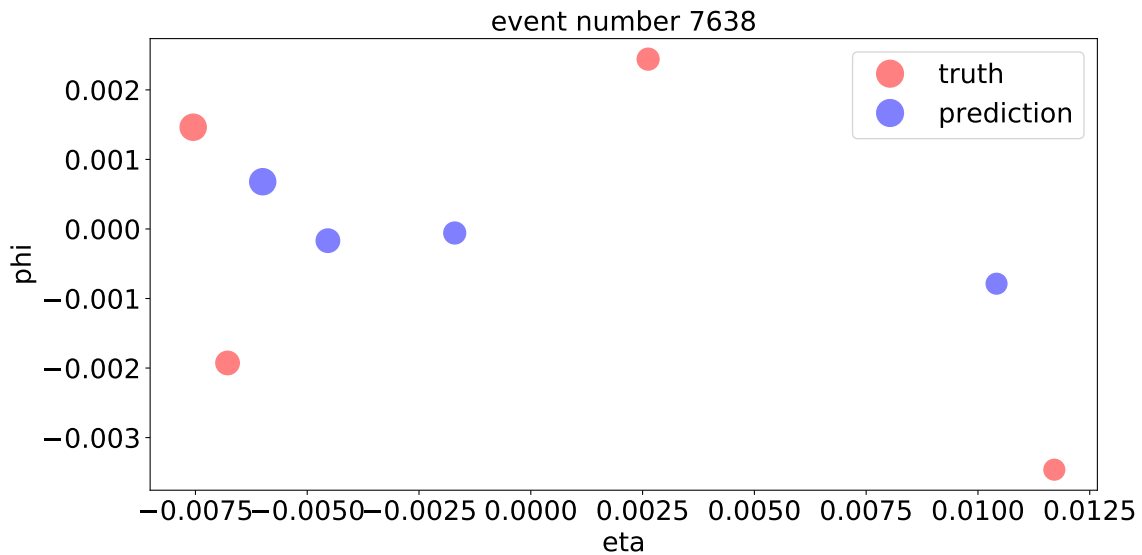


Figure C.5: Reconstruction image for the same event as in the previous chapter, on the left for the current model, and on the right for the previous one

In figure C.5 you see the difficulty of comparing a reconstruction image. Is this one is reconstructed better than figure C.2? Sure both are not very good reconstructions, but clearly saying that the new one is better seems not possible. One reason why we think this new reconstruction is better, is because it shows one feature, that will be a common feature in the next models: The reconstruction has a lower angular width than the input (this is explained in chapters 3.3.1 and 4.5).

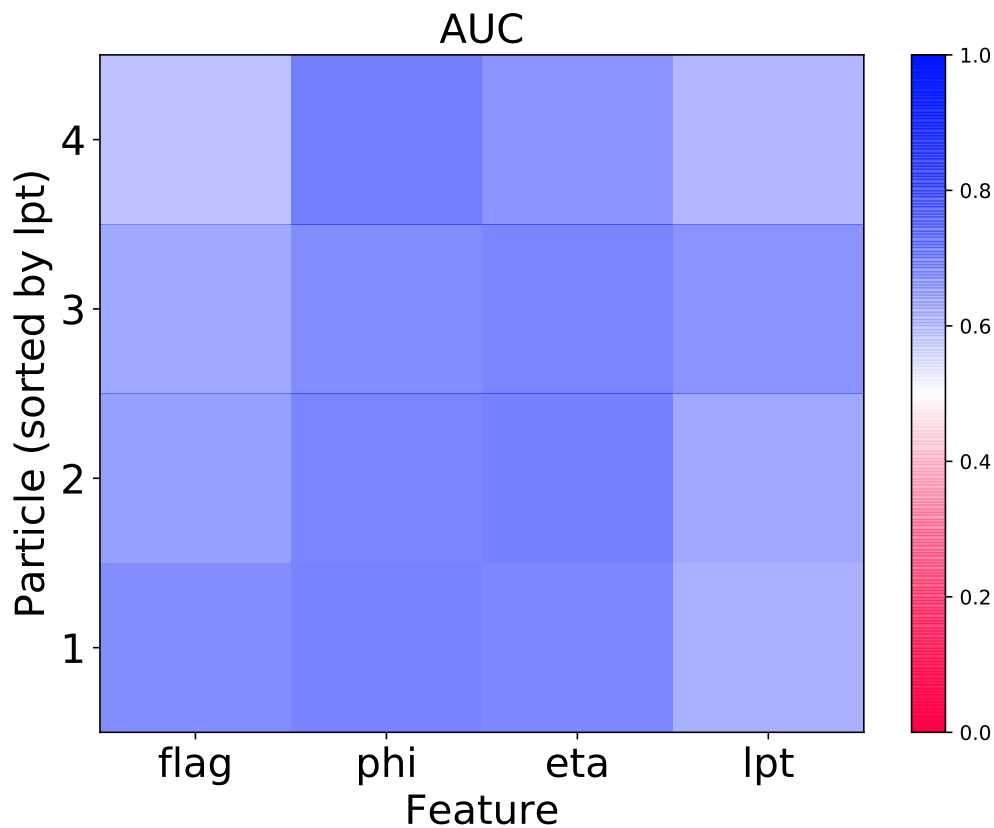


Figure C.6: AUC feature map for this model

Finally, the AUC Feature map in figure C.6 is way better than before: The angular columns are still the most important part of the reconstruction, but the flag column is no longer showing numerical problems<sup>123</sup>, and even the transverse momentum can be used to compare jets<sup>124</sup>.

## C.4 Improving autoencoder even further?

Referenced in: [2.3] [C.5]

There are some algorithmically changes that we thought of, that will be tested in this chapter.

### C.4.1 Physical intuition behind the encoding algorithm

Referenced in: [4.2] [C.5]

The usual encoding algorithm could be seen, as inverting a particle decay: Taking for example a simple two particle decay: On the graph, you could understand it as some function, which is making 1 node into two nodes. And as you can find the original particle by some function of the resulting particles, you can use an original particle with some additional attributes<sup>125</sup> to reconstruct the new particles from it. This might suggest that this kind of autoencoder is optimal for particle physics, but this setup is even more useful as it does not simply cut away additional information, and the physical problem is actually not that optimal, since the number of particles in each decay does not only have to be constant, but also known before in every compression step. Also, particles don't decay in steps: It could well be, that the initial particle decays into two particles, of which only one continuous to decay further. That being said, the optimal encoding algorithm, that we would like to be able to write (appendix C.5), would be solve this, and thus have even more physical intuition.

### C.4.2 Better encoding

Referenced in: [4.2] [9.1]

Since writing this much more advanced graph abstraction algorithm, would have taken very much time, let us focus first on a bit more simple better encoding algorithm: The current encoding basically completely ignores any graph information. After any compression stage the whole graph has to be relearned, and connections only indirectly<sup>126</sup> affect the corresponding feature vectors. Why not use the graph a bit more? Here we suggest that using a function of the original graph as the compressed graph might be a good idea: When compressing  $n$  vectors, you can see the adjacency matrix as a matrix of matrices, and the only task you need to solve, is how to extract some form of this initial global matrix. This is done here, by applying a function to each submatrix. We try out setting this function to be the mean, the maximum or the minimum of the original connections and compare them with or without rounding each entry to be one or zero to the usual graph compression. With the rounding you can see those options as setting a connection to exist when more original connections exist than don't, when at least one connection exist, or when all connections exist. Values in the following of  $-1$  are networks that resulted NAN losses.

The data we compare it on here, includes all the stuff we implement over the remaining chapters, which is why there is an oneoff AUC in those tables(see chapter 7), and also why the quality is generally worse (see chapter 5).

<sup>123</sup>This also suggests, that flag is no longer just a flag input value, that is copied until here. Sadly it took until 7.1, for us to notice what this means.

<sup>124</sup>The transverse momentum alone would reach an AUC of 0.78 and flag would reach 0.69.

<sup>125</sup>Like what it decays into (also ignoring uncertainties for now).

<sup>126</sup>Through the preceding graph update steps.

	mean loss	loss std	n	auc	oneoff auc
comparison	0.309	0.048	5	0.511	0.635
rounded min	0.366	0.037	5	0.533	0.623
rounded max	0.388	0.042	6	0.563	0.484
rounded mean	0.342	0.036	5	0.56	0.556
min	0.371	0.025	6	0.559	0.562
max	0.372	0.025	6	0.565	0.541
mean	0.351	0.04	5	0.54	0.486

Table C.1: Quality differences for different encoder with a learnable handling of the feature vectors

As you see, this is generally a bad idea, as our comparison network (the one explained in the chapter 7.3 beats them basically everywhere. Finding this here, does not mean that the corresponding layers are useless, as they simply could not be good for the data at hand, but only says that we should not use them here, which is why they are not used for the rest of this thesis, but still included in grapa.

Another thing we test, is how well a learnable parameter transformation, as used before, works, compared to also applying a function (mean, max, min) to each feature vector

	function	mean loss	loss std	n	auc	oneoff auc
comparison		0.309	0.048	5	0.511	0.635
rounded min	mean	0.366	0.016	4	0.656	0.472
rounded max	mean	0.333	−1	1	0.656	0.549
rounded mean	mean	0.372	0.016	4	0.656	0.542
min	mean	0.37	0.007	4	0.656	0.503
max	mean	0.362	0.002	3	0.654	0.55
mean	mean	0.363	0.01	4	0.655	0.505
rounded mean	min	0.296	0.022	5	0.579	0.543
rounded mean	max	−1	−1	−1	−1	−1

Table C.2: Quality differences for different encoder with a fixed function

Evaluating this test series is not as easy as the last one. In the loss, the comparison network is better than all other layers, excluding rounded means with a min function, but the oneoff AUC is worse at this model, and at all other ones. That being said, when setting the function to be the mean, the usual AUC beats the oneoff one a bit. We choose not to use this, because the higher loss means that worse autoencoder produce these results, and a consistent AUC score, that is not reproduced by the oneoff AUC suggests trivial features, which suggest no generality and invertibility.

### C.4.3 Better decoding

Referenced in: [4.3] [9.1]

Also, the decoder, does not use the graph structure completely. So we try to replace the abstraction with a constant learnable graph, by an abstraction with a graph that is not constant. The problem here, is that the tensorproduct introduced in 4.1.1 does not work for a product of one graph with multiple graphs. The main difficulty lies in finding out how to work with the nondiagonal terms: Consider again adjacency matrices of adjacency matrices: When each feature vector becomes a vector of feature vectors, also each entry in the adjacency matrix becomes a new matrix. These matrices, multiplied with the original entry would result

in a tensor product, when the new matrices would always be the same, but this is what we want to change. Finding now the diagonal matrices can be left to a learnable function of the feature vector, but for the off diagonal matrices, we have two suggestions: The first, graph like decompressor, define those matrices as functions of the two corresponding diagonal matrices. Here we compare a product, a sum and those rounded versions and or not only to the abstraction with a constant graph, but also to the second suggestion: param like decompressor: instead of the diagonal matrices being functions of a feature vector, every submatrix is a learnable function of its two corresponding original feature vectors.

	mean loss	loss std	n	auc	oneoff auc
comparison	0.309	0.048	5	0.511	0.635
product	0.265	0.019	4	0.568	0.557
sum	0.305	0.026	5	0.566	0.514
or	0.404	0.248	18	0.562	0.502
and	0.354	0.074	22	0.587	−1

Table C.3: Quality differences for different graph like decoder

This table looks much more interesting: the product based graph like decoder is able to improve the loss compared to the comparison model, while also being very reproducible. And even though the oneoff AUC is worse, this does not seem enough to exclude this model from further consideration. That being said, we will see later, that graph like decoder (and also param like ones) have a much less strong relation between the loss and the AUC, making them very good autoencoder, but not so good anomaly detectors.

We can also look at the way, the original graph is combined with the newly generated graph. Instead of using a product, we can also use a sum, or again round the result to an or<sup>127</sup> or an and. Since the combination with a constant graph is not very interesting, we use param like decompression for the practical results:

	mean loss	loss std	n	auc	oneoff auc
comparison	0.309	0.048	5	0.511	0.635
product	0.28	0.037	4	0.553	0.522
sum	0.3	0.024	4	0.549	0.526
or	−1	−1	10	−1	−1
and	0.348	0.038	16	0.631	0.546

Table C.4: Quality differences for different param like decoder

Here we see the same thing as in table C.3 Products seem to be a good idea, beating the comparison network in loss, but not in AUC.

Finally, since we now have a little adjacency matrix, and a list of feature vectors for each original feature vector, we can apply a graph update step on those subgraphs, to hopefully enhance the decompression by mixing the decompression of the adjacency matrix and the feature vectors. This was already enabled in all previous compression and decompression tests, but is tested here again on param like decompression:

This shows basically no difference, so since we like the subgraph actions from a theoretical standpoint (and because they seem to help the network converge a little faster), we keep them enabled, when we use more advanced networks.

<sup>127</sup>In practice we expect the product to be virtually identical to the or, since the inputs are either 1 or 0.

	mean loss	loss std	n	auc	oneoff auc
yes	0.28	0.037	4	0.553	0.522
no	0.277	0.0335	3	0.571	0.528

Table C.5: Quality difference for either running learnable sub graph updates or not

## C.5 The compression algorithm that we wish we would be able to write

Referenced in: [4.2] [9.1] [C.4.3]

Our approach to encoding and decoding, as discussed in chapter 4 might work fairly well, but if not already the more complicated approaches discussed in appendix C.4 would work worse than our more simple solution, and maybe even more importantly, the best possible algorithm we can think of, would not be nearly impossible to write, improving the compression and decompression would be fascinating. This is why we want to briefly talk about how you could do this: Consider the graph in figure C.7.

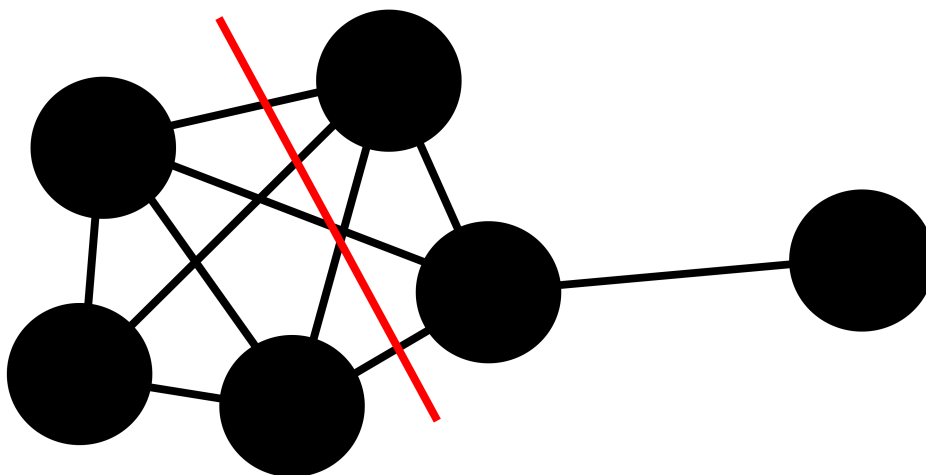


Figure C.7: A sample 6 node graph splitted using our algorithm

In the current algorithm, we would try to separate this 6 node graph into for example 2 3 node graphs, but how useful would that be? Sure you could separate them by one of their axis values, but would not the separation in figure C.8 be much more useful?

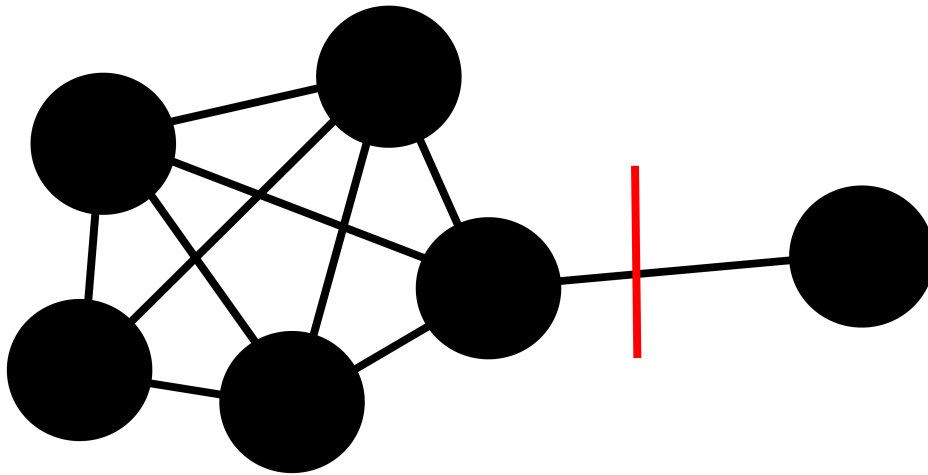


Figure C.8: A sample 6 node graph splitted how we would like to split it

To create this graph, you would use a more graph based pooling algorithm. The algorithm MinCut<sup>128</sup>, would be really hard to write, since we would want it to result in graphs of different size, whatever algorithm is then applied to the subgraphs compressing them, would be able to handle different numbers of nodes. This also means that the decompression algorithm, would return either different sizes of graphs, which would not only be hard to write but also be difficult to handle considering that this could not work in a graph like manner, and might be at least limited in a param like manner, or would be combined into a graph bigger than the original one (while also limiting subgraph sizes). And considering that already sorting graph nodes is worthy of discussion (see appendix A.3) and sorting is much easier than cutting your graph into an usable size, this is definitely not trivial to implement.

So why do it? Appendix C.4.1 gives you some more physical intuition why this might work better on jets, but even when this would not work, choosing a better encoding and decoding algorithm still might be very useful as a graph pooling and a graph generating algorithm (see appendices F.2 and F.4 respectively)

<sup>128</sup>From [12], mincut tries to separate graphs by the least number of graph lines.

## D More problems while writing a graph autoencoder

### D.1 Choosing the right compression size

Referenced in: [B.1] [D.2]

One problem of each autoencoder anomaly detection approach is that the size of the latent space is arbitrary. Mixed approaches like in chapter 7 solve this by working with every compression size, as long as the autoencoder is not trivial. But not using mixed approaches this is not so trivial.

There are two things to consider:

- The compression size for the autoencoder in chapter 4 we choose by optimizing their AUC value. And as we have discussed in detail for example in chapter 5, this is a bad idea: This resulted in the compression size being way too low to contain the whole information, and thus no useful feature. Generally you can say that you don't want your latent space to be too small, as you lose information in this case.
- The other thing you don't want, is a compression size that is too big. A perfect reconstruction is invariant under changing the training data, and you won't be able to use this Autoencoder at all. We also noticed that there seems to be a minimal compression size for a normalized network to be invertible. This minimal size we chose in chapter 6.

We are quite happy with our choice of compression size, but it is still a good idea to raster every possible compression size. This is done for a specific unnormalized network in appendix B.1.

### D.2 Building identities out of graphs

Referenced in: [4.1.1] [6.1.2] [9.1] [C.1.2]

An optimal autoencoder should be equivalent to the network with the compression size set to the input size. The problem here is, that this trivial model does not necessarily reproduce its input perfectly. As described in chapter 4.1, the graph update step is given by

$$f(n_j^k \cdot A_k^i \cdot x_i + s_j^i \cdot x_i) \quad (\text{D.1})$$

and this kind of update step is not always invertible through another step. To see this, let us first ignore the activation function as  $f(x) = x$  and let us use a fixed size. Given 3 nodes of 2 features each, let the adjacency matrix and thus the graph be fixed to be:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad (\text{D.2})$$

While the 2x2 matrices are general, we can use the tensor product from chapter 4.1.1 to convert them, corresponding to converting the 2 dimensional feature vector in a one dimensional one, into a corresponding matrix that can be multiplied to this new one dimensional feature vector. This matrix will then be given by

$$\begin{bmatrix} s_{00} & s_{01} & n_{00} & n_{01} & 0 & 0 \\ s_{10} & s_{11} & n_{10} & n_{11} & 0 & 0 \\ n_{00} & n_{01} & s_{00} & s_{01} & n_{00} & n_{01} \\ n_{10} & n_{11} & s_{10} & s_{11} & n_{10} & n_{11} \\ 0 & 0 & n_{00} & n_{01} & s_{00} & s_{01} \\ 0 & 0 & n_{10} & n_{11} & s_{10} & s_{11} \end{bmatrix} \quad (\text{D.3})$$

This example we already use in chapter 4.1.1, but the point here is now, that, to be able to create a good identity, the inverse of this matrix has to be of the same structure, but the inverse of this matrix is not of the same form. This can be most easily seen by considering the last element of the first and second line of the resulting identity. We call the corresponding neighbor interaction matrix of the second step here  $m$ , while the self interaction does not appear<sup>129</sup>

$$0 = m_{01} \cdot n_{00} + m_{11} \cdot n_{01} \quad (\text{D.4})$$

$$0 = m_{01} \cdot n_{10} + m_{11} \cdot n_{11} \quad (\text{D.5})$$

which can only be solved for

$$\frac{n_{00}}{n_{01}} = \frac{n_{10}}{n_{11}} \quad (\text{D.6})$$

but since  $n$  is given, the matrix cannot be invertible<sup>130</sup>.

You could ask yourself if this is actually a problem, since even though two nonactivated update steps cannot invert themselves, but surely a bunch of update steps are invertible together. We also assumed the adjacency matrix to be the same, which does not actually has to be the case. And even if not, since the compression size is not the same as the input size, the problem is anyway different. Sadly this is not something that we are able to easily calculate, but what we can do, is test this experimentally. As shown above, any graph update step can be rewritten as a product with a specific matrix. This allows us to create an inverse update step, that is equivalent to the normal one, except for a numerical inverse of the update matrix and train networks using those inverse update steps to decompress our data (You could ask yourself if the update matrix is invertible, and in general it is not (a trivial example might be no neighbour interaction and a noninvertible self interaction<sup>131</sup>), but in practice this is a problem that can be controlled: It happens that the function used (`tf.linalg.inv`) fails, but this is rare, can be controlled by the initialiser of those matrices, and even if it fails, the documentation states that this function might<sup>132</sup> just return noise instead of showing errors. And considering that having a matrix that is not invertible requires each parameter to be exactly tuned, this can be ignored by the parameters constantly changing. A bigger problem, and the reason, why we do not use these invertible matrices in each decompression phase are those matrices that are nearly not invertible (have a determinant very close to zero). Since the determinants of the inverses of those matrices are huge, they can amplify noise and thus confuse the minimization algorithm. In practice that means that a network that once has reached a quite low loss, can have quite a bigger loss after a couple more training steps). Sadly this results in less stable training, which is enough for us to not use inverse update layers.

So finally we see those invertibility problems as another kind of loss. Next to allowing only for  $n$  out of  $m$  features to be used, these  $n$  features have to work around the structure of the graph. This means, that comparable autoencoder of a non graph type work better for smaller compression size. This might seem terrible, but we think it only means, that each compression size for a graph network is equivalent to a smaller one for a nongraph network, and thus, some compression sizes close to the maximum are impossible. Finally, you could even see this as a benefit for graph autoencoder, since choosing the right compression size is not a trivial task (see chapter D.1), and this gives you kind of a regulator, saving you from choosing a too high one. Also, please note, that the ability for the autoencoder to reconstruct data, does not imply anything concerning its effectiveness as classifier

<sup>129</sup>This is to expect, since the self interaction part has to obviously invertible (at least in most cases).

<sup>130</sup>You could argue, that  $n$  is learnable, but this expects a bit much from the learning algorithm. More explicitly reducing the possible neighbour interactions into a small subset reduces the possibilities of the learning algorithm and the worth of the graph drastically.

<sup>131</sup>Aka a no self interaction.

<sup>132</sup>Yes, the documentation is not very precise what happens in such a case.

### D.3 Is permutation invariance good or bad?

Referenced in: [C.1.2]

Permutation invariance is a central feature of any (good) graph network, but is this actually a benefit? On the one hand, you can use permutation invariant data in a more natural way, but on the other one, you also have to find a way of making your whole algorithm permutation invariant. The only point this becomes critical in this thesis, is in the decompression algorithm. Since the compression algorithm changes the order of nodes, but the decompression algorithm not, might compare reconstructions in a suboptimal way. We see two ways of solving this. One would be to make your loss function permutation invariant. But when we have tried this, it did seem to hurt the overall performance. The easier alternative is simply to sort each particle by one of their variables. We discuss this in appendix A.3.

### D.4 Why use graph autoencoder

Referenced in: [D.5.1]

From our experience, Graph autoencoder have some clear advantages over classical autoencoder. This does not mean, that they don't have problems(as discussed in appendix D.5), or even that those benefits usually outweigh the problems, but at least that there might be situations, in which choosing a graph autoencoder would be a good choice. The most obvious situation, in which graph autoencoder should be chosen, is if by their input data has the form of a graph: That means multiple vectors (or more general objects), with some relational information between them, that should not be ignored. This can also be beneficial for variable number of vectors, or when a permutation symmetry between the input vectors in a set is expected. Another benefit is the separation in multiple similarly handled vectors. This similar handling does not only keep the number of trainable parameters low, and thus makes overfitting hard<sup>133</sup>, but also makes interpreting the output easier, since when every attribute of the same kind is treated the same, there are not many differences between the qualities of different particles, but more between different attributes. Also and probably most useful, these shared parameters<sup>134</sup> keep the number of needed training samples quite low: Even though more training sample cannot really hurt the network quality, we could without problems, reduce the training size down by more than two orders of magnitude from 600k to 5k (see appendix B.5), and it seems to be possible to reduce these training size even further, allowing us to build useful networks with only  $O(1)$  training samples (see appendix F.3). Finally, changing the graph setup layer can change the whole meaning of a graph layer, transforming a layer that handles physical distance into one that cares only about momenta. This can allow for variable metric setups, that can iteratively focus on whatever is important at the current position.

### D.5 Why not to use graph autoencoder

Referenced in: [9.1] [D.4]

Given the reasons for why to use graph autoencoder in chapter D.4, here we want to list some reasons why not to use autoencoder in general.

#### D.5.1 Reproducing vs classifying quality

When we started working on anomaly detection, Autoencoder seemed like are quite a good idea, a simple way to differentiate between things that are known, and things that are not known, while still giving you a way of testing how good your models are trained, without needing anything else but background data by just evaluating the quality of the autoencoder. It stands

<sup>133</sup>Or in the models we trained basically impossible.

<sup>134</sup>The low count and the demanded similarity.

to reason that a bad autoencoder did not understand the background data in any way, that can be used to differentiate it from the signal data, but this is basically just an assumption, and so after having a bit more experience encoding and classifying data, and especially when we now have another method of separating data, we first want to spend some time evaluating this hypothesis. To do this, let's look first at the loss of the autoencoder: Since it is basically just the difference between input and output(simplifying chapter 4.5 a bit), it is a measurement about how good the autoencoder reproduces whatever we put into it<sup>135</sup>, and so by just calculating the loss on background data, we have a measure for the quality of the autoencoder. So basically we want to see a strong falling correlation between the loss of our network and the AUC score<sup>136</sup>, do we see this? We show here the loss in training against the decision quality of the corresponding network in this training step. Please note that this is exactly what we want, since the correlation we want here, would mean, that in training, a classifier gets continuously better<sup>137138139</sup>.

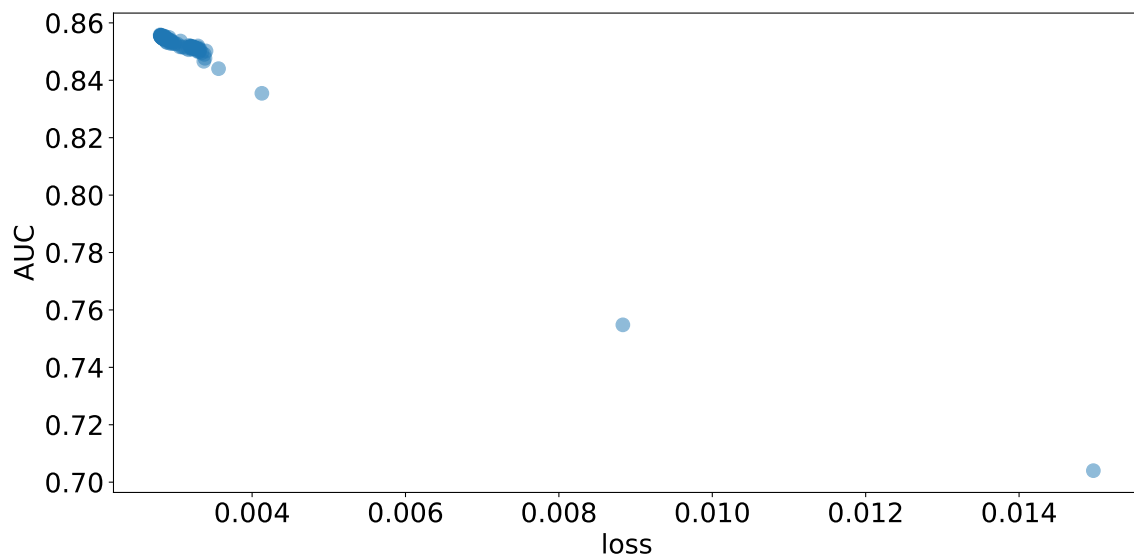


Figure D.1: A simple old AUC by epoch plot for a unnormalized network with thus focus on angular data

At first glance almost all networks, at least since we consider them working autoencoder, are monotonously falling like in figure D.1, but there some side marks: Most importantly is this one of those networks that we trained just to have a high AUC, and thus is a network that basically just compares angles to zero(see chapter 4). This does not mean that this image is useless, as it shows, that at least a networks finds out that a way less useful feature should be ignored, but we should look at the AUC by epoch of a good network:

<sup>135</sup>This is also a bit of a simplification, since the normalization of the data matters a lot, but we will come back to this later.

<sup>136</sup>The lower the loss, the higher the AUC, since we train on QCD jets.

<sup>137</sup>Yes you could argue, that it is enough when the highest AUC is reached at the lowest loss, but in practice this is not enough because the network does not always reach the same point.

<sup>138</sup>It should be noted, that a network cancels the training when it does not improve for a certain number of epochs, so in theory you do not know if there may not even be a better classifier at a way later epoch, but with worse loss. This is usually assumed to be false, since overfitting defines the rest of training, but since we do not see basically any overfitting, this might not be so easily ignored.

<sup>139</sup>We should also note, that this is an analysis that we did only for a small fraction of all networks, since evaluating the quality of hunderts of networks takes considerable time, often even more than training itself. This is also why we change this method later, to consider only those epochs in which the network improves its loss.

As you see, this relation is basically the same as before, except for two differences: The new networks are worse, and we do have way fewer epochs. This is since we stopped calculating the AUC for each epoch, but now only calculate it for each epoch in which the network improved<sup>140</sup> More interestingly, please note a peculiarity in the preceding images: As you see the relations are almost linear. This is not necessarily always the case, consider figure D.2.

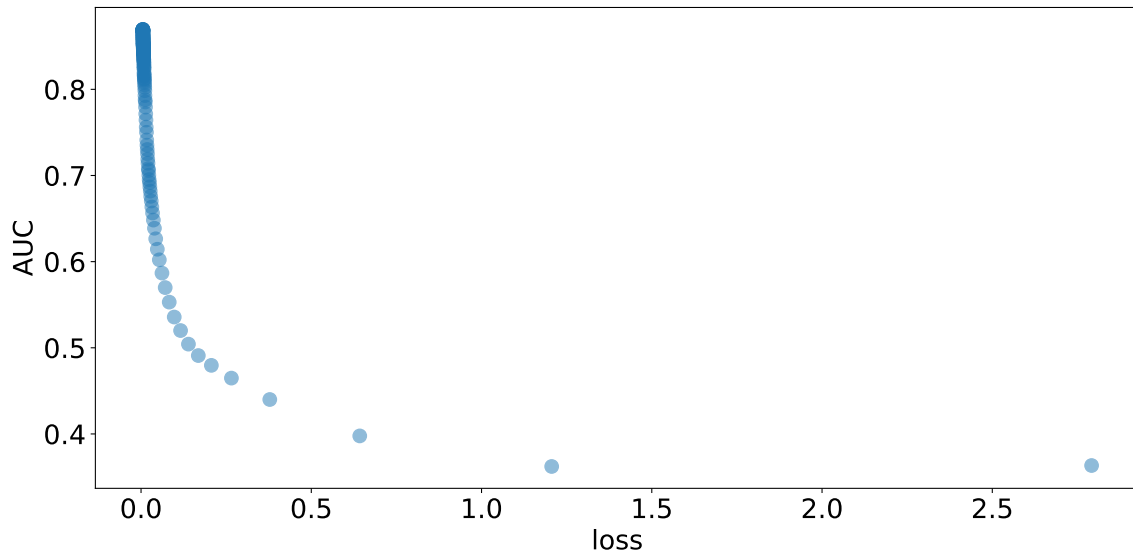


Figure D.2: AUC as function of the losses without normalization using a trivial decompressor.

It should be noted, that this image is still of the bad kind, focussing mostly on the angular part, but as you see, this image still shows a strictly falling relation, while this time it seems to be exponential in nature (something like  $c - e^{-d \cdot x}$ ). You could ask yourself what is better? The exponential one might be limited in its quality, but is also easier to saturate. And this is most likely a feature of our network architecture, since you get this curve when you replace the decoder with a more trivial one. Interestingly the quality in the linear seeming case cannot be linear, since the AUC has to always be below 1, so you could ask yourself how this curve continuous. The obvious first assumption would be that both curves are exponential in nature, but we are not able to saturate the capacities of a nontrivially decoding network. As logically as this seems, testing this is a bit of a different story: Since apparently our networks don't reach the necessary quality to saturate themselves, this is hard, if not impossible to test. Which is why give up testing just one autoencoder, either by stopping to test just one network, or by testing other kind of classifiers, and as you will see, both suggest a similar, again falling quality curve for little losses: Let's start with multiple networks, instead of plotting parts of one network, we plot the result of multiple networks:

<sup>140</sup>This we did for computational reasons, not only to save time, but also disc space. And even though this makes our reasoning a bit less accurate, this should be a reasonable compromise.

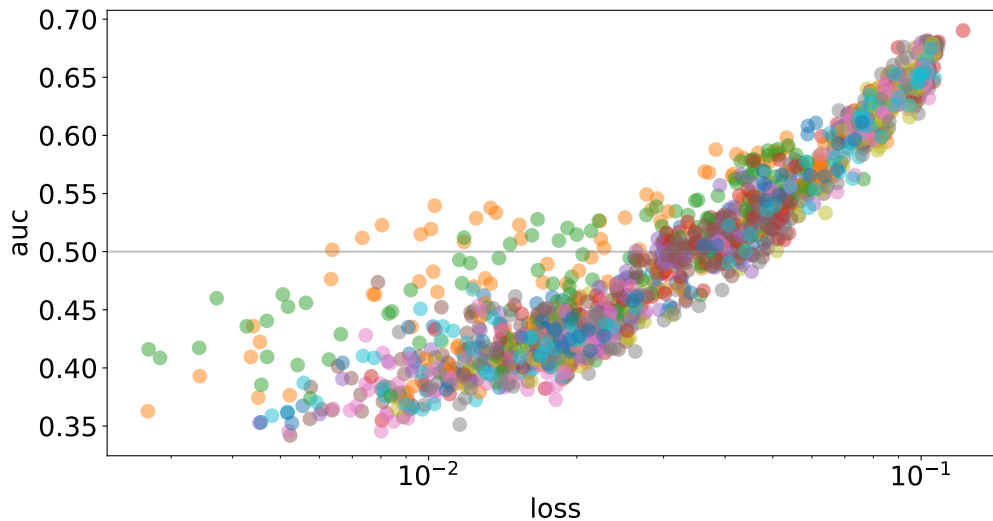


Figure D.3: Comparison of multiple network, with other decompressors. green represents graph like and orange parameter like decomposition. Compare this to figure 6.5 in chapter 6.1

As you see in figure D.3, this relation is basically the same. Since the problem in this step was reproducibility, you have a lot of different random network qualities, but, as already mentioned in chapter 5, there is a strong relation, that would be linear if the x-axis would be linear, and that is growing, since it is trained on top data, and in our definitions, the optimal AUC would now be 0 instead of 1. The more interesting part is now those part at tiny loss, that seems to deviate from the relation. These are two network types, that have a more complicated decoder, and as you see: They are definitely way better autoencoder, but are also less good classifier. Then consider chapter 7.1, and oneoff networks that show basically the same relation(see figure D.4).

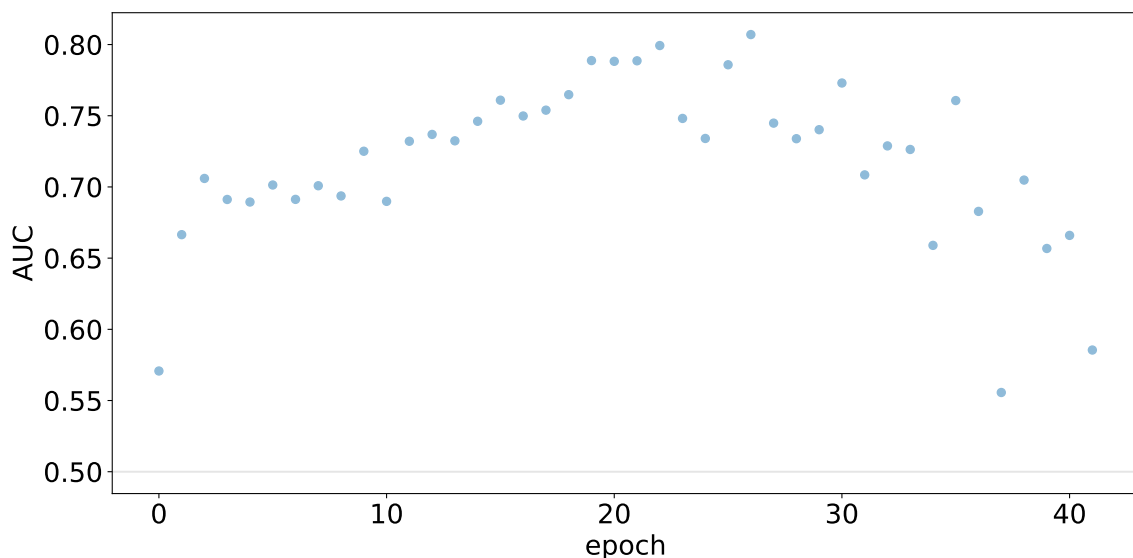


Figure D.4: AUC by epoch for oneoff showing a growing relation, that at some point starts to fall again

So lets us assume, that the quality falls of again at some point, why could that be? One reason might be, that signal and background are similar in certain features. Consider the following network: you feed it one particle only, but not only the 4 momentum, but also the

mass: An autoencoder with the right compression size would learn to reconstruct the mass from the momentum 4 vector, probably more than it could ever find patterns in the 4 momentum itself. And if you consider that top quarks decay similarly as QCD quarks, there are certainly similarities that an autoencoder should not focus on. This explains mostly why oneoff networks decrease in quality, since they just focus on one feature, but this is also a problem we can solve with a mixed approach, as shown in chapter 7. On the other hand, this might explain why autoencoder this effect less, since they combine features, instead of relying on only one. That being said, this combination of features might be the real problem: while talking about feature combination, we assumed that a saturated classifier, is an optimal classifier, but this is not actually the case. Consider the c addition explained in chapter 5.1.4 and tested in chapter E.4: Any feature that is less useful has a bigger influence, that has to be compensated by a power 3 in this width, and since the leading pt particle is easier to reconstruct<sup>141</sup> than for example the particle with the 7th highest pt, certain parts of the reconstruction are more or less useful, but the combination makes no difference between particles, so their combination will not be optimal, and thus the combination might only be saturated with bad combination factors. In fact we can look at this, by looking at the partial networks from chapters 5.1 and 6.2.1<sup>142</sup> when we don't add them together with their optimal factors, but with each factor being 1

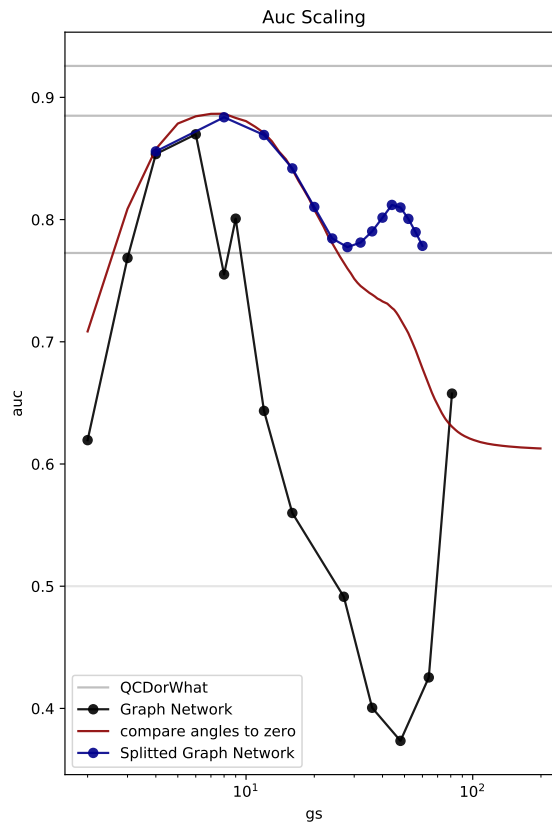


Figure D.5: Partial network combinations, AUC as function of the graph size(gs) comparing a combination with equal weights to one with a loss power of 3

<sup>141</sup>Meaning the leading pt particle is less random.

<sup>142</sup>We should point out, that this is not entirely the same, since instead of adding particles, we here add bunches of 4 particles each, but we would not expect single particles to add differently than particle bunches, and training a graph on a single particle does not exactly make the best use of their relational setup.

You see first in figure D.5, that the combination of equal factors reaches less good AUC, while still converging against a certain AUC Value. You might also notice that the equal factor one does only decrease at some point, but this happens, because as seen in chapter 5.1.4 There is a zone of factors, in which adding them together is not optimal, while still increasing the AUC score, so a decreasing classifier combination is just a more extreme version of nonoptimal combinations, that appears when two classifiers are "too different". Also we just talked about different particles but same features, how about different features? How to you combine features with different meanings? C addition only works, since we assume that similar good reconstructed features have a similar width, and when you give this up, the width is no longer a measure only for quality but also for certain attributes of the feature itself<sup>143</sup>, and any algebraic combination becomes less useful, and also learnable approaches are tricky: Even though a perfect combination (a combination is a normalization here), might result in the maximal AUC score, finding it can lead to nearly optimal maxima (as seen in 5.1.4). Another idea might be a learnable combination but here there is still the question of focus: When we talk about c addition, we also assume that there is a certain choice of things the network should focus on, and anything the network does not care to reproduce, is less accurate information<sup>144</sup>, but by making this focus basically learnable there is also a problem that you might call normalizing and this is absolutely nontrivial. If you ignore normalizing your combination vector, the network will just learn zeros into it (a loss is always minimal if you compare zeros to zeros), but even when you assert that for example  $|c| = 1$ , this does not necessarily solve your problem, since the network might focus entirely on the smallest feature, and even if you assert that the size of each feature is the same, it might focus on the feature that is easiest to reconstruct, and thus it might still be nonoptimal<sup>145</sup>. As a final note in this subchapter: Please consider that oneoff networks should in theory solve all of this combination problems, since they work on minimizing their width only, and the optimal combination for a minimal width can be assumed to be the same as the combination for maximum AUC (see appendices E.3 and E.4), but as chapter 7 and 8 show, they have their own problems.

---

<sup>143</sup>Consider the following network: it has some features and the same features times 4 as input (with some noise, that the autoencoder ignores), sure it will find out that all of its features are twice to reconstruct, and it will just learn to set the second batch of features to the first batch times 4, but how to combine those features now? If we just let the mse do this, the second batch is way more important, but this is unlogical, the best addition (ignoring noise for now) would be the first feature\*4 + the second feature, and we can get it by simply normating the features in a certain way, but now consider the following: what if the noise of the second one has a different size than the noise of the first one? Then we would have to consider this, first multiply the first feature by 4 and then use c addition, but to do this, we need to know the multiplicative factor, and in general we wont.

<sup>144</sup>There is also the inverse effect: when the network just copies a certain information, there is less decision quality in this feature potentially failing c addition.

<sup>145</sup>We think you might be able to do solve this, by playing reconstruction complexity against size using c addition, but we have not tried this, and we expect it to be quite finicky.

## E Understanding Oneoff networks with more precision

### E.1 Other algorithms

Referenced in: [7.2]

Since oneoff networks seem to have potential, that is just not used that well on jets, you could ask yourself if other one-class methods work better. So this chapter serves as an introduction into several of those classical algorithms for finding signal events after training on background events, as well as a reasoning why this is not the case. The field these algorithms belong to, is called one class learning.

#### E.1.1 Support vector machines

Classically, SVMs are used to differentiate two sets of data points, by drawing lines between them, so that they are completely separated. Instead of using deep learning, this problem could be solved analytically, even with the extension no longer requiring only lines, but a learnable transformation of a line, and thus allowing SVMs to not only work on linearly separable data. This might be more powerful, but still cannot handle every possible data distribution, and this problem stays the same for the one class learning version. Here you draw some shape (usually a circle) with some transformation (an ellipse) around your given background data, in a way that minimizes the volume. This restricted amount of possible shapes can be useful, keeping the SVM from overfitting, but it also only allows it to learn certain distributions, so distributions like figure E.1 could not ever be learned.

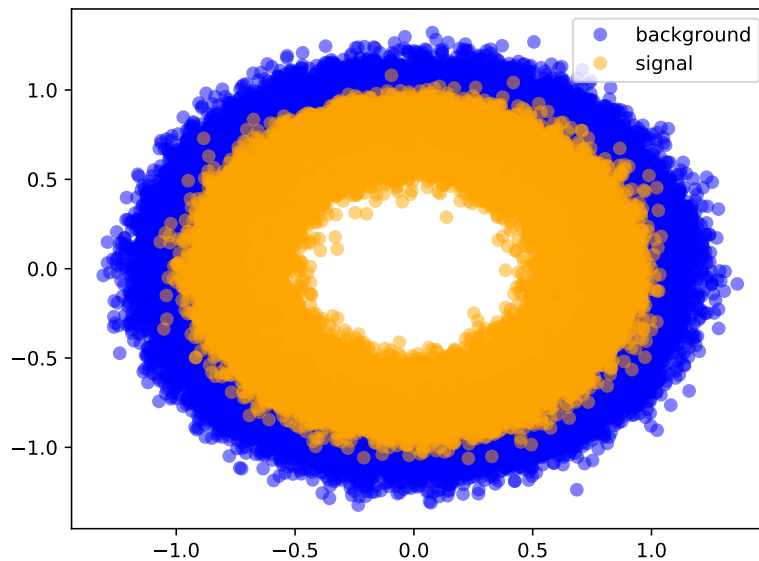


Figure E.1: A simple example of a shape, that an SVM cannot differentiate

This is a problem, since a shape like this, could be the result of a simple rotational symmetry, which are not uncommon in physics, so as you might expect, training an SVM<sup>146</sup> on QCD jets to find top jets does not result in any useful results (an AUC of 0.53).

<sup>146</sup>Implementent in sklearn.

### E.1.2 K nearest neighbours

Another usually quite useful algorithm is also an extension of a supervised task: Given two classes of vectors, you can classify each new point, by looking at the class of the vector that is closest to it, or at the mean of the classes of the  $k$  Nearest vectors to it. This you can extend to the one class case, by setting the loss of one vector to be the mean of the differences to its  $k$  Nearest neighbors. Since those known points are only background events, you can expect an abnormal event to have a higher loss, while background events are probably more similar to already known background events. The problem comes from its ability to overfit. This can easily be understood in the supervised case, since single weird background cases can lead to a region in which no signal can be detected. You could say that autoencoder focus on the distribution of events, Support vector machines focus on the outliers of their distribution, while  $k$  nearest neighbour focusses on the whole volume, which means, it could solve the above distribution E.1, with a quite good AUC of 0.89 for  $k = 1$  and 0.96 for  $k = 100$ <sup>147</sup>, and in the jet case, this algorithm does better, reaching an AUC of about 0.37 (on top), but it is still limited by the curse of dimensionality: One class learning algorithm usually work better on low dimensional inputs than on high dimensional one, and here this can be understood quite easily, since the volume of possible vectors grows exponentially with the dimension, while the number of training samples won't change too much, making the difference between each of the background events statistically bigger.

### E.1.3 Isolation forests

An isolation forest[35] works quite different to the algorithms explained before. Instead of defining what a normal event looks like, this algorithm tries to isolate anomalies. It does this, by randomly classifying points into a tree: Given some attributes, it picks a random one and a point at which to separate the data by. By iterating this procedure, you build a tree, in which abnormal data usually is separated easier than the normal data, which means you can use the depth of a position in the tree as separator. That being said, this algorithm might be interesting, but still does not work very well, possible since it is also not immune to the curse of dimensionality, and so the result on QCD vs top is also only an AUC of 0.502.

---

<sup>147</sup>There is some overlay between signal and background in the image, which limits the AUC.

## E.2 Different algorithms for latent space training

Referenced in: [2.2] [7.2]

Here we test different one class algorithms on one autoencoder that trains on the first 4 particles of top jets and tries to find QCD jets as a signal. For comparison, simply looking at the loss of this network, you get an AUC score of about 0.377 and the best (the lowest, since we train on top jets here) AUC score we have seen for an autoencoder is about 0.25. We choose this one, since its reconstruction seems to be quite accurate<sup>148</sup> (see figure E.2), meaning that most information about the jet has to be still contained in the feature space

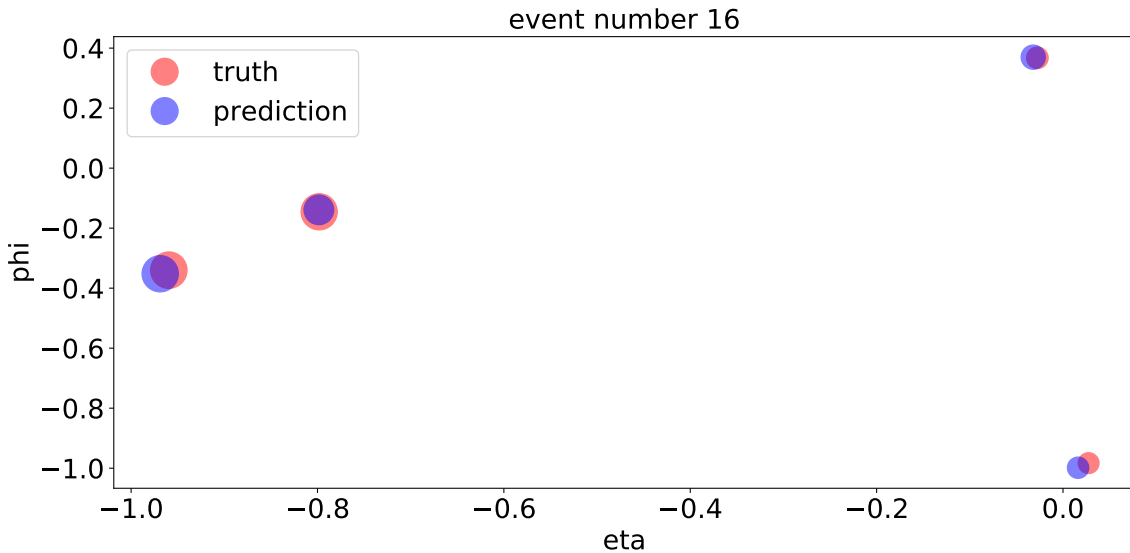


Figure E.2: Reconstruction image for the comparison algorithm

### E.2.1 SVM

An SVM works better on the compressed space, now reaching an AUC of 0.434, but even though this is definitely an improvement, it is still worse than just using the autoencoder

### E.2.2 Isolation forest

Also the isolation forest improves, now reaching an AUC of 0.377, so it is at least as good as simply using the autoencoder.

### E.2.3 k nearest neighbour

K nearest neighbor is the first algorithm that improves over simply using the autoencoder loss, reaching an AUC of 0.307, even though it should be noted, that this is for  $k = 1$  and gets worse for any higher  $k$ . Also, this is still worse than our best autoencoder.

### E.2.4 Oneoff

Oneoffs seem to be the way to go here, and will thus be used exclusively for this thesis: One network reached about 0.247 with an error of 0.005, already beating all our autoencoder, and by combining multiple ones, you can reach an AUC of about 0.2 being quite good.

<sup>148</sup>At the point of testing one of the most accurate algorithms.

### E.3 Oneoff math

Referenced in: [7.1.1] [D.5.1] [E.4]

To give you a better understanding on how oneoff networks work, let's talk about the math behind this idea a bit, especially how this kind of networks should handle multiple kinds of information. To do this, let us consider a simple model: Each feature is build out of two gaussian distributions, the first distribution describes the training/background data, and thus has a mean of 1 and some width  $\sigma_1$  and the second one describes the signal data, it has a mean of  $\mu$  and a width of  $\sigma_2$ . This means the decision quality of this feature can be described by  $s = \frac{|\mu-1|}{\sqrt{\sigma_1^2 + \sigma_2^2}}$ . The higher  $s$  is, the bigger the difference between both peaks, and the better the separation and thus the higher the AUC. This might remind the reader of the math considered in the chapter about c addition (5.1.4), and it actually concludes, that by considering how to combine two background, the math is exactly same as for c addition: Given two distributions, with width  $s_1$  and  $s_2$  and mean values of 1, you can combine both distributions into one distribution with smaller width. This width of the distribution  $\frac{c \cdot d_2 + d_1}{c+1}$  is given by  $\frac{\sqrt{c^2 \cdot s_2^2 + s_1^2}}{c+1}$  while the mean is still 1. This function is exactly the same, as was minimized to find the combination with the best AUC in chapter 5.1.4. This might suggest, that the resulting combination of an oneoff network is the combination with the highest possible AUC. Sadly this is simply not so easy: The problem are the assumptions made in the chapter about c addition(5.1.4) We set the distance between the background and the signal peak to be constant, which results in the width of the distributions to be the only important thing to consider, when combining two distributions. This is fine, when considering features of similar kind, since you can assume their distributions to be similar, but does not anymore here: And when you assume this distance to be more or less random, the calculation becomes a bit more complicated. In fact, you can assume, that any other possible peak could be some sort of signal data, that you want to exclude (see chapter E.4). So consider the following model: Given a background peak around 1 width a certain width  $s_1$ , and an improvement of this peak, being more focussed with a width  $s_2 \leq s_1$ : Which peak is more probable to separate a random signal from the background? Since the second peak is less wide, it is less probable for a signal peak to overlap it and thus probably results in a higher AUC score (figure E.3)<sup>149</sup>

<sup>149</sup>To be more precise, you optimize the background peak by improving on the function that generates it, this clearly also changes the signal peak, but does this in a more or less random manner. It could improve the AUC, but could also hurt it. The point is, that this change is random, and thus on average, optimizing an oneoff network might be useful. We do this more precisely in chapter E.4.

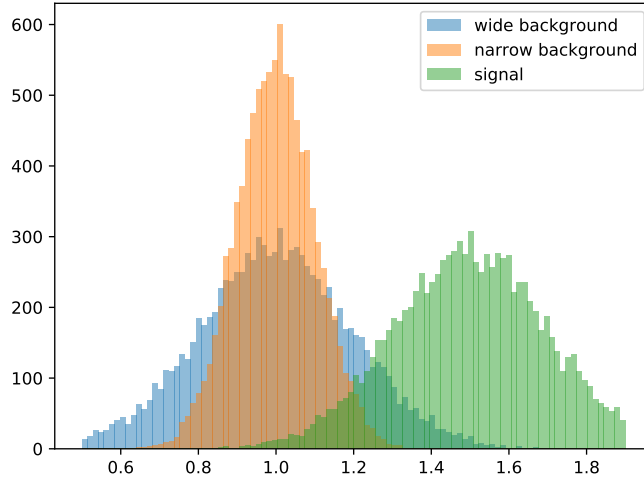


Figure E.3: Two different widths of a background peak, resulting in different overlappings to the signal peak

This combination of two improving methods, C addition for similar features, and statistical improvement for asimilar features, is why we think, that oneoff networks might work well, but there are two caveats we need to talk about here: First, this optimization does not help at all, when the distance between the background the signal peak is zero, since when the oneoff network focuses on something that is the same in background and signal, making a distribution less wide, only results in both getting smaller. In practice this becomes only a big problem, when you have trivial inputs, consider the case of the autoencoder discussed at the beginning of the chapter: If you change the normalization to have a trivial 1 in one of the other outputs, you lose all decision power of the flag variables. A possible solution to this problem will be discussed in the next chapter 7. Another caveat that should be mentioned, is the loss you use for training an oneoff network. The first choice might be to just minimize  $(x - 1)^2$ , but if you try this, you notice that the resulting mean is not 1 but smaller than 1. To understand this, consider the following model: given a gaussian peak  $i$  with a width  $\sigma$  and a mean of 1, which distribution  $i \cdot \mu$ , with a constant  $\mu$ , minimizes  $(i \cdot \mu - 1)^2$ ? This loss can be written<sup>150</sup> as

$$\text{gauss}((x - 1)^2, 1 - \mu, \mu \cdot \sigma) \quad (\text{E.1})$$

where we write  $\text{gauss}(a, \mu, \sigma)$  as the application of a gaussian kernel with mean  $\mu$  and width  $\sigma$  around  $a$ . This function has the same minima as<sup>151</sup>

$$\text{gauss}(x^2, 1 - \mu, \mu \cdot \sigma) = \mu^2 \cdot s^2 + (1 - \mu)^2 \quad (\text{E.2})$$

This function is minimal at  $\mu = \frac{1}{s^2 + 1}$ . Since this value is not always 1, training on this loss, while comparing the signal peak to 1, does not work. You can fix this, by simply training on a different loss  $(\text{mean}(x) - 1)^2 + \text{std}^2(x)$ <sup>152</sup> works quite well, or you can just read the mean of the output distribution, by subtracting the mean of the background peak, instead of 1. Both methods work with similarly good results, even though their output is not always the same. In the following basically always readjusted means are used.

<sup>150</sup>When approximating the number of training samples as infinite.

<sup>151</sup>Since the constant part does not change the minima, and the linear part is zero by symmetry.

<sup>152</sup>Where  $\text{mean}(x)$  returns the mean of the input distribution, while  $\text{std}^2(x)$  returns its variance.

## E.4 Self improving oneoff networks

Referenced in: [4.6.4] [5.1.4] [5.2.1] [7.1.1] [7.2] [8] [9.1] [D.5.1] [E.3]

We justified oneoff networks before (see E.3) by showing that the factor needed to combine two double gaussian features is the same, when we talk about minimizing the the width of the first peak, or when we use  $c$  addition to find the maximum AUC value. That might seem great, but there is one assumption, that we kind of glossed over a bit in the chapter about  $c$  addition: The difference between both peaks, is not necessarily defined by the size of the first peak. So here we want to go into detail about why this is not necessarily a problem, and what consequences can follow from breaking this assumption. The way we try to understand this, is by looking at width vs AUC plots, of different  $c$  values combining random gaussian double peaks, that have each a random width between 0 and 2, while having fixed means of 0 and 1 each<sup>153</sup>. After simulating a lot of random distributions, three nontrivial<sup>154</sup> classes seem to emerge. Those are shown in figure E.4.

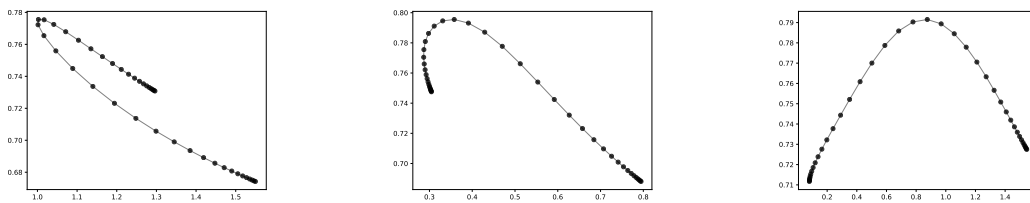


Figure E.4: The three kinds of simulated AUC by loss behaviours

As you see, the first relation is pretty much perfect: the lower the width of the first combined resulting peak, the higher the AUC value is. This is the class we want, and we would get if the assumption would be true. Sadly this is not the only possible result and the second class is not that optimal: These are distributions of suboptimal combination, where the lowest loss, does not result in the highest AUC, but at least into some value that is close to the expected optimum. This class appears in different levels of accuracy, reaching from distributions, that are nearly indifferable from the optimal case, to some, that are definitely not good. Finally, the third class, contains combinations that are completely suboptimal: The optimal AUC value is reached at a basically terrible loss, and by decreasing the loss, the AUC becomes bad again. These are what you might call traps: a very bad classifier is hidden behind a small initial distribution. You can easily see why this cannot ever be filtered out, by considering the case of a trivial feature, that is just always (for signal and background) 1: the oneoff network will focus entirely on it, since it can reach a loss that is exactly zero, ignoring every feature that would be better at classification, and thus reach a useless classification score, and by looking only at the background distribution, there is nothing you can do<sup>155</sup>. Please note, that in this case, the autoencoder would actually solve the problem: since the feature is trivial, it will be filtered out, and thus cannot be learned from the oneoff network. Combine this with the fact that, from a quick simulation, this case does not seem to appear too commonly

and you can suggest that this will not be a problem.

But to test this, we have to work on actual data, so this is the loss vs AUC relation for a network trained on the compressed space of top jets, trying to find QCD as signal. Please note, that there is a huge difference to the simple case of optimally adding gaussian double peaks: Firstly, there is an unknown<sup>156</sup> number of features being combined. This won't change too

<sup>153</sup>This is still general, since translation and scale invariance give us 2 degrees of freedom per doublepeak.

<sup>154</sup>With trivial we mean relations that are defined by at least one doublepeak with an AUC of 1.

<sup>155</sup>Except for using a different algorithm, for example a SVM.

<sup>156</sup>Unknown, as we cannot really find out, how many informations the network uses to classify a feature.

Class	Number
optimal	12
close to optimal	6
very close to optimal	5
between close to optimal and terrible	5
terrible	5

Table E.1: Distribution of types of loss vs AUC plots for random gaussian double peaks

much, since the quadratic addition of  $n$  features can be understood as the quadratic addition of one feature with the quadratic addition of  $n - 1$  features, but might be valuable to keep in mind. Secondly, instead of looking at different values of a  $c$  factor, combining the peaks, we only look those  $c$  values, the network considers at the end of each epoch: That means, we could overlook an optimal AUC in the middle of an epoch, or even miss a good classifier never considered by the network.

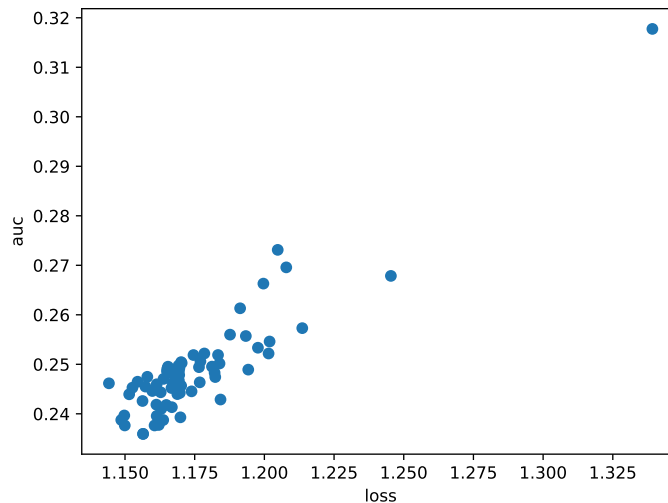


Figure E.5: AUC as function of the loss for a oneoff network

In figure E.5, the AUC seems to fall with the width of the first distribution (this is what we want, since we train on top jets), but not reaching the true optimal value. Comparing this to the theoretical expectation is not that easy: the most obvious reason why this does not matter might be the inaccuracy of the AUC values: the value might not be optimal, but is very close to the optimal value, while the AUC seems to fluctuate about more than the expected value. More interesting is the relation at less than optimal AUC values. Sadly there are not that many points here. This correlates to the fact, that oneoff networks gain most of their progress in their first epochs (if not epoch), but the points that we see, seem to fit quite nicely to one, if not two lines, building the tails of the theoretically expected relations for an at least close to optimal case. That being said, even ignoring the theoretical expectation, and the possibility of another even better combination, this is a relation that validates our training procedure, suggests that the initial autoencoder works at removing traps, and might even suggest, that one of the reasons, multiple oneoff networks combined are better than only one, comes from

---

Since 12 normalised networks usually don't set trained parameters to zero, and any gaussian peak can usually help reduce the width of a peak (Central limit theorem), it is actually reasonable to assume, that the whole compressed space, as a transformation of a here 9 dimensional feature vector, is used.

the fact, that multiple distributions reduce the noise in the AUC loss relation, and thus gain statistically better AUC values<sup>157</sup>.

### E.4.1 Oneoff outside of physics

Referenced in: [7.1.1]

This apparently unused potential led us to try them out on more classical evaluation datasets, and we found a paper[47], that not only works fairly similar to oneoff networks<sup>158</sup>, but also evaluates them quite thoroughly on MNIST<sup>159</sup>. One algorithm they test their algorithm against is based on autoencoders while another uses GANs, and they constantly outperform them. We test here oneoffs on the following task: Given drawings of the number 7, how well can you detect other numbers. They provide also the results from assuming every other number to be the background, but here we focus on 7 for now (the other numbers are very similar), since it seems not too easy, while also not being too hard of a task. They reach an AUC score of 0.946 with an error of 0.009, while oneoffs reach a quality of 0.914 with an error of 0.018. You could see this, and think that again, they have potential, but they are definitely worse than the reference paper, but this would ignore one fact: Their approach does take the whole datavector to retrieve its loss, while our only takes some part, and by retraining the oneoff network, they do not predict the exact same thing<sup>160</sup>. This means that it should be easy to combine multiple runs into one good classifier, and the math for this (see chapter 5.1.4) is even easier here, since every network could reach the same quality, you can set  $c = 1$  and just add each value of  $|x - 1|$  together. If you do this with enough reruns<sup>161</sup>, the AUC converges against a value of 0.981, beating the comparison paper, and thus showing the true potential of oneoff networks for one class learning.

<sup>157</sup>Even though it should be noted, that this cannot be the only effect, since this combination usually results in about an increase of 5%, while this only seems to account for at most 1%.

<sup>158</sup>They use something called a support vector machine, which is probably most easily described as an algorithm that draws a circle like shape around the known data points, and classifies everything inside the shape to be background, and everything outside to be signal. Their main idea is to make the shape to be learnable in a deep way. So the main difference to oneoff networks is the fact that here there is a certain region, with the smallest possible size, optimal to be in for the background events, while in oneoff networks the only values that are optimal are exactly one.

<sup>159</sup>MNIST is a set of handwritten digits, that is often used to test new algorithms [32].

<sup>160</sup>On average there is a correlation of about 0.6 between each retraining.

<sup>161</sup>We used here 25 runs.

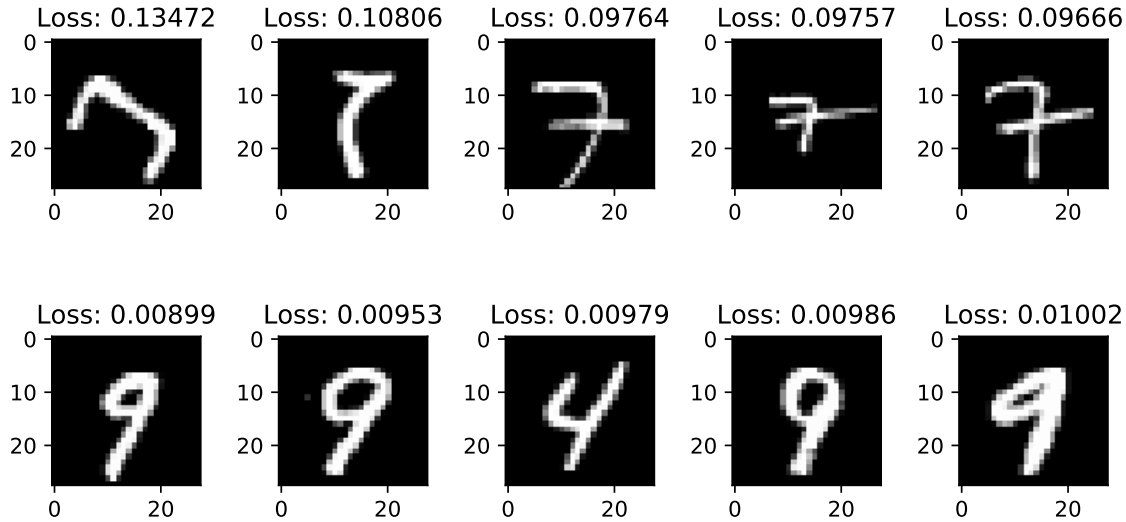


Figure E.6: On the top: The 5 least 7 like 7th in the training set. On the bottom: The 5 most 7 like not 7th in the evaluation sample. Our favorite image is that in the lower right corner, as you can clearly see that it is a 9, while also allowing you to see how it can be interpreted as a 7.

We show some classification examples in figure E.6.

#### E.4.2 Physical interpretability for oneoff networks

Referenced in: [7.1.1] [A.1]

Another example, why oneoff networks might be quite useful, comes from our experiments to understand them more. Instead of constructing arbitrary features by utilizing deep networks, the algorithm used here only combines input features linearly. The data we work on here is provided by cern open data as two lepton events from the 2010 datasets. Momentum 4 vectors of muons[38] as background and of electrons[37] as signals. These 4 vectors are squared with a linear metric, reducing it into one dimension, that is evaluated to minimize  $(|g_{\mu\nu} \cdot p^\mu \cdot p^\nu| - 1)^2$ . This results in the network learning the following metric

	$e$	$p_1$	$p_2$	$p_3$
$e$	-0.4997	0.0011	-0.0002	0.0002
$p_1$	0.0011	0.5069	0.0014	-0.0008
$p_2$	-0.0002	0.0014	0.493	-0.0006
$p_3$	0.0002	-0.0008	-0.0006	0.4998

Table E.2: Learned metrik values of oneoff networks trained on muon events

As you see, the result is very similar to a Minkowski metric: The nondiagonal parts are zero in the range of numerical uncertainty (and symmetric for 5 digits behind the commata), the signs are randomly this way, because of the absolute value in the loss function and the absolute value of the diagonal parts scales the resulting expected output of 1 that the loss expects. Other than this, this simple network is able to understand itself, that characterizing a particle is best done through what we call its mass. That being said, the AUC score is not optimal, only reaching 0.5988, but we can improve this, by assuming the metric to be strictly diagonal, which results in a learned metric of E.3.

$e$	$p_1$	$p_2$	$p_3$
1.4198	-1.413	-1.4151	-1.4197

Table E.3: Learned metrik values for a diagonal metrik oneoff networks trained on muon events

As you see, this still results in a minkowski metric like result. This time with a flipped sign, and a different scale, which is just a feature of the implementation. Most importantly, this simplified metric definition, including less noise, results in a much higher AUC value of 0.8007<sup>162</sup>

## E.5 How an oneoff network can become noninvertible

The easiest model for understanding the oneoff width is something like  $\sqrt{|x - \text{mean}(x)|^2 + \text{std}^2(x)}$ . And while the means usually match the training data, the standard deviation can be of any size. So training on a dataset and comparing it to another dataset with the same mean and less width, results in a noninvertible network. This is nothing we can do anything about, and an effect that is the same when we talk about autoencoder classifier, and is even less probable here, as we try to minimize the width, making it less probable that there is a distribution with lower width. Still this can happen (you can see the frequency in chapter 8.3), but here we want to mention one effect, that is even worse: antiinvertibility: if trained on a, b has lower loss, and if trained on b, a has lower loss. This is an ultra rare effect, as we have only observed it once (or multiple times if you think of the statistical invertibility of chapter 8.1), and an effect that cannot happen just with an autoencoder, so how does this happen? In general, an oneoff network should not be able to do this, as if one feature has a certain width in a, and a lower feature in b, you should be able to pick the same feature in b resulting in b finding a more complicated, except for the case in which there is another feature in b with lower width, but this would also mean, that the width of the second feature in a would be bigger than of the first feature, since else it would have been chosen, resulting again in b finding a more complicated. Or in math: given  $f_1^b \leq f_1^a$  the network is not antiinvertible unless  $f_2^b \leq f_1^b$ , but since  $f_1^a \leq f_2^a$  it has also to be true that  $f_2^b \leq f_2^a$  so no network can be antiinvertible<sup>163</sup>. That being said, since we use autoencoder in the front, it can happen, that a feature of the first autoencoder just does not exist in the second one, thus breaking the logical chain, and making antiinvertible networks possible. We only ever saw a single event doing this, and it was a network working on ldm data (see chapter 8.1). Ldm data is hard to differentiate at best, making noninvertible networks much more likely (in fact, as seen in chapter 8.3, all noninvertible oneoffs in this thesis are trained on ldm data), and by trying to scale using dense networks, at a node number of 25 we got an antiinvertible network. This is shown in figure E.7.

<sup>162</sup>You could ask yourself, why we use muons as background events: This is because the relative uncertainty of each electron mass value is much bigger, since the mass is more than two orders of magnitude smaller. Training a (only diagonal) network like this, still results in a minkowski like metrik (-0.0058,0.0043,0.0043,0.0058), but the AUC value is way worse reaching only 0.5003 as the expected mean value has way less physical meaning.

<sup>163</sup>We simplify here a tiny bit, since you could mix two features, but this does not change the math.

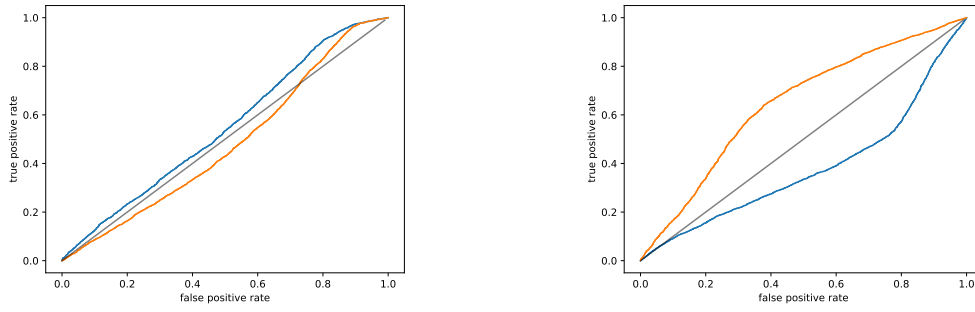


Figure E.7: Antiinvertibility for ldm data, left using the autoencoder loss, and on right using oneoff networks

Interestingly is the separation quality here much better (even if reversed), as we will be able to do in chapter 8.1 with node sizes of 4. This we interpret that this difference is basically just the jet size, as the number of nodes show a difference between both ldm datapoints in figure E.8.

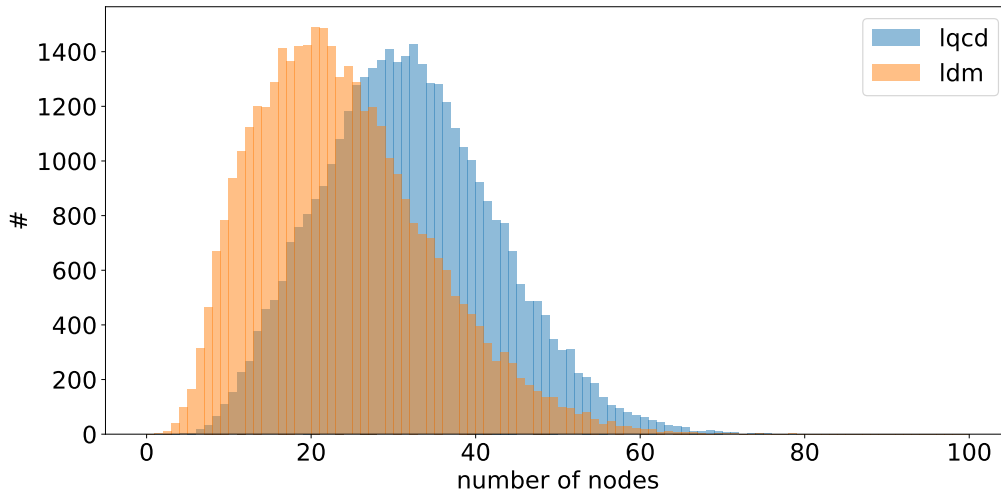


Figure E.8: Number of jet particles for ldm jets

and a network with fewer nodes (we tried 9 and 16, sizes that don't show many zero particles<sup>164</sup>) does not show any real difference between both datasets. 36 nodes show a difference, but while not being invertible, this dataset is at least not antiinvertible, showing how rare those networks are.

<sup>164</sup>Zero padded particles to be precise, for more information see chapter 3.2.

## E.6 Why c addition might not be perfect

Referenced in: [5.1.4]

. C addition (chapter 5.1.4) is a very simple model. And while it seems to match our assumption approximately, there is a difference that can be explained by the assumption we made.

- We assume the loss to correlate to the signal peak, which in generally can not be assumed.
- We also assume no correlation between those features, which is definitely not the case.
- And there are some other minor assumptions, like gaussian like peaks<sup>165</sup>.
- As well as an assumption on the equivalence of each trained part, while at later points, Jets with particles that do not exist start to become important, and even though they will be reconstructed easier, this events definitely are of an a bit different kind, possibly requiring different combinatorics.

---

<sup>165</sup>To be precise: Peaks of the shape of an strictly monotonous transformation applied to a gaussian.

## F Other usecases for grapa

Referenced in: [4.3] [9] [9.1] [B.5]

This chapter is based on 4 graph autoencoder applications, that were originally written as easy tutorials for the documentation of our graph layers<sup>166</sup>. And, even though there are some quite interesting insights about graph autoencoder to be found here, this also means, that this appendix can be skipped, without losing too much information. Finally, this also means, that each of those applications are not optimized in any way, and maybe not even completely though through, since their main goal is just to be a quick explanation of the code structure and maybe to be some inspiration on what is possible using a graph autoencoder.

### F.1 Abnormal account detection for social networks

Referenced in: [2.3] [5.1.3] [F.3.3]

Social networks provide data that is naturally described by graphs<sup>167</sup>, so by training a network on them, with the hope of finding abnormal users, we not only get a new possible use case for graph autoencoder, but also an example code for a network that does not generate its own graph.

The corresponding tutorial can be found at <https://grapa.readthedocs.io/en/latest/nets.html>

#### F.1.1 Datageneration

Data generation is often the most time-consuming part of a new neural network, and it would not be different here. So to save some time, we just generate a sample social network. This allows you to ignore privacy settings<sup>168</sup>, simplifies the problem a bit<sup>169</sup>, and allows you to clearly define the anomalous data points. That being said, this also means, that we could tweak the data in every possible way to make the results arbitrarily good, which is also why this is the only subchapter that works with self generated data. This generated network consists of 5000 randomly generated users with 4 attributes (A constant 1 (flag),  $a$ :an integer between 1 and 3,  $b$ :an integer either 0 or 1 as well as a normal distributed value that depends on  $a$ <sup>170</sup>). The corresponding connections are generated the following way: each connection has a probability, that depends on the difference in the person vector(a factor  $e^{-|x_i - x_j|}$ ) and on the difference in the node index(another factor  $e^{(-0.1) \cdot |i - j|}$ ). This means, that more similar persons are connected more closely, and that friends of friends are more probably friends. Now we guess on average 5 connections for each person, with respect to the given probabilities, or 2 for the alternative data points. We choose these anomalies, since defining less used accounts as signals allows us later to show a benefit of oneoff networks. Now for each person in this network, we only look at the local surrounding of this person. This is done, by taking only the connection of the friends, or friends of friends of this person into account. This generates a bunch of smaller graphs, that we can now feed into the autoencoder<sup>171</sup>, but for simplicity we cut a bit on the size of those

<sup>166</sup><https://grapa.readthedocs.io/en/latest/> .

<sup>167</sup>If you let users be nodes, while friendships provide the edges.

<sup>168</sup>You might not know everything about every user: you would need to decide how to handle a friend about whom you do not know some critical information.

<sup>169</sup>Since an usual facebook user has a lot of information, and often enough hundrets of friends.

<sup>170</sup>A normal distributed value with mean 0 and standard deviation 1 added to  $2^a/16$  times another normal distribution with mean 1 and standard deviation 0.1. This is done just to have some relation between the elements.

<sup>171</sup>You could ask yourself if this reusing of nodes does not result in a lot of overfitting (by learning the nodes themselves), but as you see below, that is not the case, possibly because of the low number of parameters in the graph autoencoder.

new graphs, as we allow for at most 70 nodes<sup>172</sup>.

### F.1.2 Training

To train this network, we use a fairly simple setup, compressing the 70 nodes once by a factor 5, resulting in 14 nodes for which we allow 12 informations each. The training curve is quite boring, showing the loss being basically the same for trainings and validation loss. Much more interestingly, is the loss distribution in figure F.1.

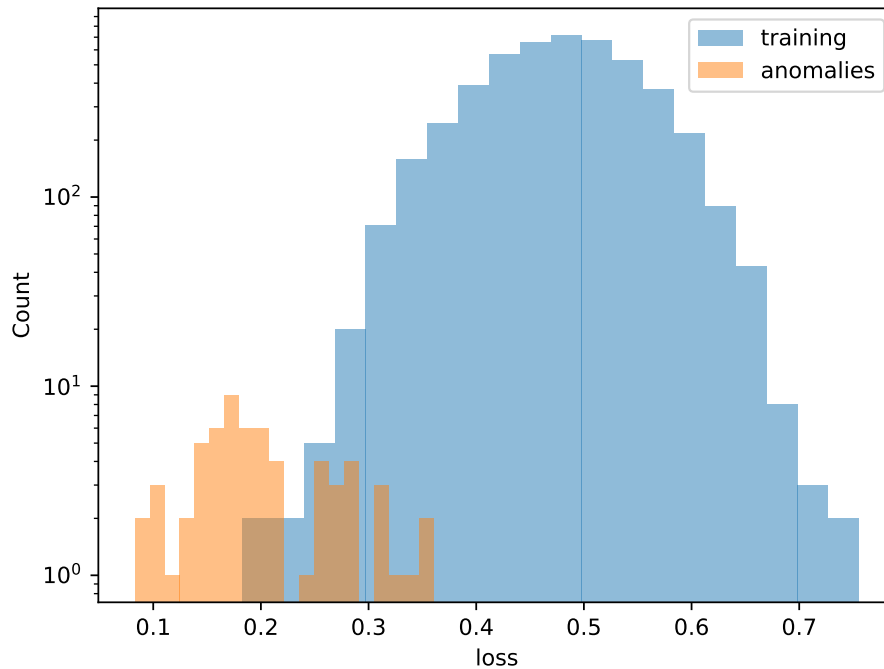


Figure F.1: Loss distribution for social networks

As you see, the reconstruction is not very good, as basically all events have a nonzero loss, but maybe even more important: there is some difference in the reconstruction of our abnormal data points. This might seem like you could use this to separate datapoints, but there is a difficulty: If you use an autoencoder to separate datasets, you assume that a dataset which the network never saw, will be reconstructed worse than a dataset that is trained on, but here the opposite is the case: the data that is abnormal is easier reconstructed<sup>173</sup>, so any separation is a bit weird: you could just look at something like  $1 - \text{loss}$ , but since you do not have any reasoning for this makes the training no longer unsupervised. Also probably only this kind of abnormal data will be reconstructed easier<sup>174</sup>, and by negating the loss, you would not get any useful separation on other data points. So what can we do? Use oneoff networks: In their easiest version, they take the mean of the training peak, and define distance as difference to this peak, which would already solve this problem, and in their deep implementation they might even improve this further. In any case, this works quite well as seen in figure F.2.

<sup>172</sup>This still keeps 0.9932 of all data points.

<sup>173</sup>This reminds of the case of nonnormalized nets trained on top jets.

<sup>174</sup>This is here probably the case, since the abnormal data is less complicated, as it contains less nodes, you might be able to handle this, by defining your loss relative to the number of nodes, but this misses the point a bit, as more easy anomalies can still exist, and we can show that oneoff networks can handle them.

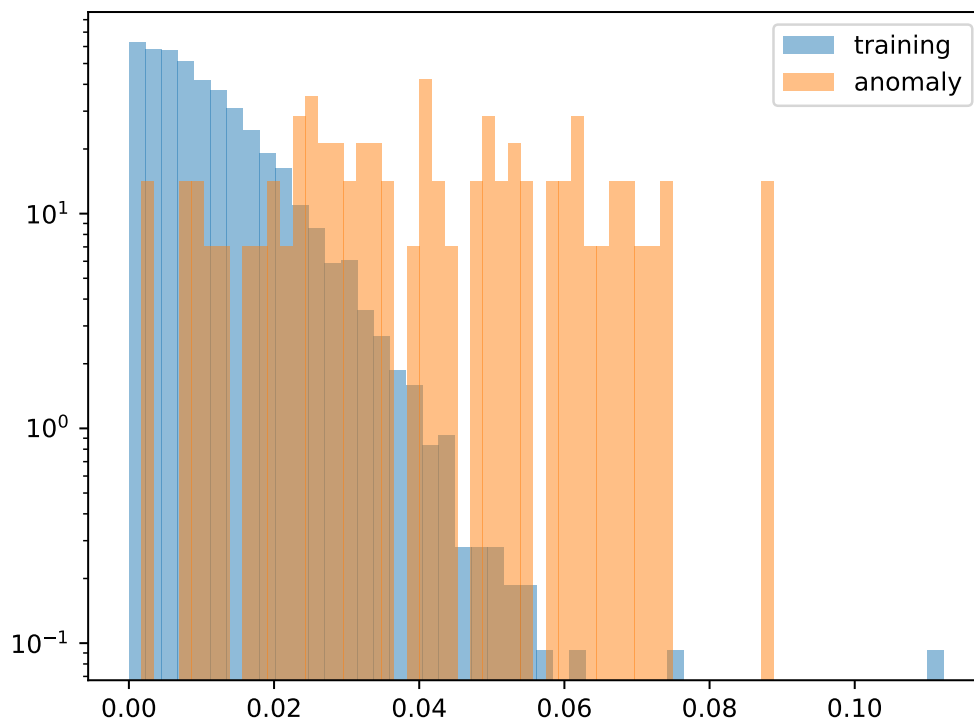


Figure F.2: Oneoff loss distribution on social networks

Please note that we dispense of using any number here, measuring how good the reconstruction is, as we could improve it arbitrarily by changing the data generation

### F.1.3 Whats next

Given these examples, you might notice, that they are not thought through completely. This is why we include these kinds of subchapter to give you some ideas on what could be improved further. The first thing you might need, is to work with more kinds of anomalous data points, and not just neglected profiles. It might also be a good idea to work on an actual social network, and if you do this, it would be interesting to just look at the users in the training set, that are reconstructed worst, as this would allow you to find abnormal users in a truly unsupervised way.

## F.2 Accelerating molecular networks through pooling

Referenced in: [2.3] [2.4] [C.5]

Our second example alternative use case works on molecules: As they are usually described only by interactions between pairs of atoms, they are well described by graphs. Here we want to use this to suggest that the compression step in a graph autoencoder can accelerate a network trying to learn a function from this molecule.

The corresponding tutorial can be found at <https://grapa.readthedocs.io/en/latest/mol.html>

### F.2.1 Datageneration

All our datapoints here are random molecules, that come from [www.chemspider.com](http://www.chemspider.com), mostly since they allow you to easily download a complete description of a molecule. This includes not only all atoms, but also suggested connections<sup>175</sup>, as well as molecular mass, here given in  $g/mol$ , which is what we use as the network output. Every atom is given by 3 spacial coordinates, that give the position relative to the other atoms in the molecule, and another attribute detailing the type of atom<sup>176</sup>. Every other information given for each atom is ignored, similar to information given about edges, except for which atoms are connected. Those data points, get filtered a lot, since fewer events do not really matter as much faulty ones, at least as long as overfitting is not a problem. First, we only allow molecules constructed entirely from the atoms H, C, O and N, which take values 1, 2, 3 and 4 respectively. Then we allow at most 50, and at least 25 molecules, of which between 10 and 25 have to be hydrogen, and check if the downloaded file is consistent<sup>177</sup>.

### F.2.2 Training

We again use a fairly simple setup, consisting only out of a handful of graph update layers, and possibly a graph compression layer, comparing its effect in figure F.3.

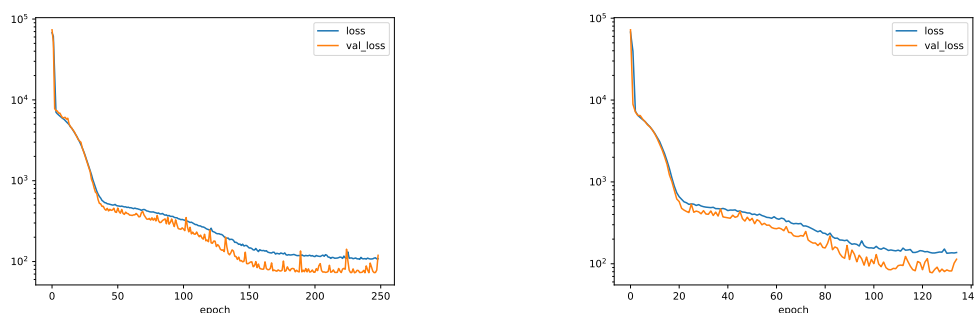


Figure F.3: Training curves on the left without compression step and on the right with one

There are two things to note here: first, since the mass can reach order of magnitude of  $1000 \cdot g/mol$ , and since the difference is squared, this can reach very high loss values at the beginning of the training. Combine this with the fact, that masses are very easy to predict, and this is why you see orders of magnitude of change in the loss function. As you also see, the change in loss is different between the compressing network, and the noncompressed version: As both networks require similar times to calculate a training epoch, the compressed

<sup>175</sup>You could also generate those connections yourself, but this would require a different algorithm instead of topK, more something that connects everything in a fixed distance (see appendix B.4.2).

<sup>176</sup>Onehot encoding this might actually be a better idea.

<sup>177</sup>The molecular formula matches the distribution of atoms.

version requires more than 100 epochs less to reach a similar result. That being said, therefore the noncompressed version reaches a slightly lower minimal loss of 73 in comparison to 78, even though it should be noted, that this difference is tiny compared to initial losses, and the compressing version has a bunch more parameters that might be able to be tweaked to change this.

### F.2.3 Whats next?

We don't want to call using a compression layer to pool graph networks generally a good idea, but if you have a network that takes a long time, trying out inserting a compression layer might be a good idea. It might also be interesting to optimize the hyperparameters of the compression layer, or even to alter the setup by for example using an abstraction layer. Finally, this is tested on a fairly easy setup, and it might be interesting to use this on a more complicated setup like ParticleNet and applies to a more complicated task.

## F.3 High level machine learning and feynman diagramms

Referenced in: [A.3] [D.4]

Machine learning and anomaly detection is usually only used on low level data. Inputs that are easily generated but time-consuming for humans to understand. But here we want to apply machine learning to highly abstracted concepts. You might ask why one would want this: One result might be something like a theory evaluation method: If you have a number of predictions, this could classify weirdness in the sense of finding predictions that don't match the rest. In the best case you could also extend theories consistently: You can generate new predictions from existing ones. You could automatically bring structure to your predictions, by looking at the compression space of an autoencoder or you could use this to simplify complicated theories. So why don't we do this? Two things come to mind: most theories can not be brought into vector form, and generating a lot of predictions is quite hard. Luckily both are solved by the graph setup: This graph structure is way more powerful, to the point that artificial intelligence research often encodes knowledge in graphs, and since overfitting has not been a problem at all here, also the low number of training samples should not matter here<sup>178</sup> Now consider feynman diagrams: As they are able to encode particle physics in a finite set of graphs, they are at the same time very high level, while also still providing  $O(1)$  samples, which should be barely enough for us to train on, and finding anomalous feynman diagrams might actually be an interesting way to solve this thesis initial idea of using graphs and autoencoding to find new physics.

The corresponding tutorial can be found at <https://grapa.readthedocs.io/en/latest/feyn.html>

### F.3.1 Data generation

Data generation for feynman diagrams means more converting data, instead of outright generating them. The problem is, that all diagrams that you find, are usually given as images, and writing an program to read every image into a diagram is absolutely nontrivial, which is why we just converted those diagramms by hand<sup>179</sup>. That being said, you could actually ask yourself, if writing an image like autoencoder to work on those images would not be much less

<sup>178</sup>There is a second price you pay, when you train on a few datapoints: Not only becomes overfitting more probable, but you also loose generality, as density fluctuations of the different kind of training samples (where these types of samples are defined by the training itself, which makes them hard to filter out) start to matter more. Sadly we cannot really change this to much.

<sup>179</sup>You could actually use a graph neural network for this, build similar to the one from the next chapter F.4.

work. And even though we would agree, we think this would also work way worse, as you could not differentiate between an image that just looks like a feynman diagram, and an image that actually represents some physical insight<sup>180</sup>. If everything looks like a feynman diagram, you can easily use the loss to differentiate those two cases, since a change in loss now definitely represents a better reconstruction in the autoencoder we will train. Also, by training on images you could again more probably see overfitting, resulting in higher needed training samples, that we don't have. We use all diagrams from [4]<sup>181</sup>, that match our filter of only SM diagrams and at most 9 lines, and represent each diagram in the following way (visualised in figure F.4): Each line becomes a node, and each two lines that meet in an edge, are connected. This might seem counterintuitive at first, as we basically switch nodes and edges, but is actually necessary, since each edge requires two nodes, and in most usual feynman diagram this is not given, as input as well as output lines, only have one edge. Then each line(node) is represented by a 14 dimensional vector, onehot<sup>182</sup> encoding the particle type (gluon,quark,lepton,muon,Higgs, W Boson, Z Boson, photon, proton,jet), 3 special boolean values encoding anti particles<sup>183</sup>, input lines, output lines and a fourteenth value that is 1 (similar to flag (see chapter 3.2)).

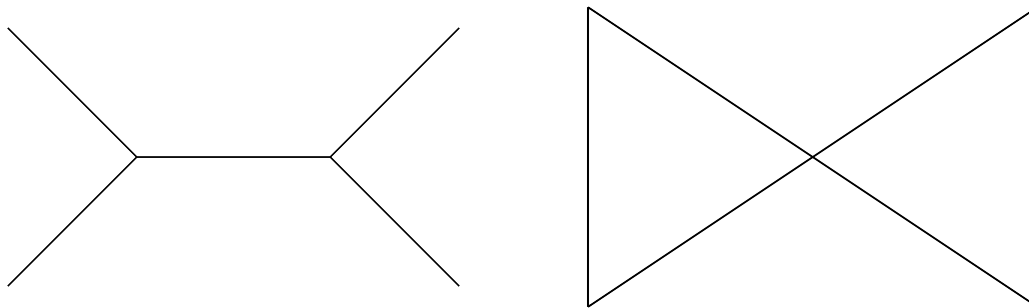


Figure F.4: Example image of the conversion used for feynman diagrams, transforming the left diagram into the right one

### F.3.2 Training

Also, here a fairly easy setup is used, but instead of the compression algorithm, we use the abstraction one, and the param like deconstruction algorithm replaces the classical one, to encode the abstraction of a factor 3 (Reducing 9 nodes into 3). Therefore, we add 3 parameters, as well as a couple more graph update steps. One thing that might be important later, is that we don't punish the resulting graph structure directly, even though the paramlike decompression algorithm should make this possible, but only indirectly through the fact that a nonsensical graph structure will worsen the quality of the update step.

<sup>180</sup>You see this quite clearly in another usecase we were thinking about: Recipes are easily generated by a text based gan, or better texts that look like recipes, but generating recipes that actually taste good is much harder, and you don't really have a way to test this(beside cooking for a long time), as your loss could also just say how much your text looks like a recipe.

<sup>181</sup>These diagrams are of fairly low order.

<sup>182</sup>Onehot encoding means encoding a number that is smaller than  $a$  by a vector of values which element  $i$  is 1 if the number is  $i$  and 0 else.

<sup>183</sup>For simplicity this variable is always zero for lines that are neither input nor output.

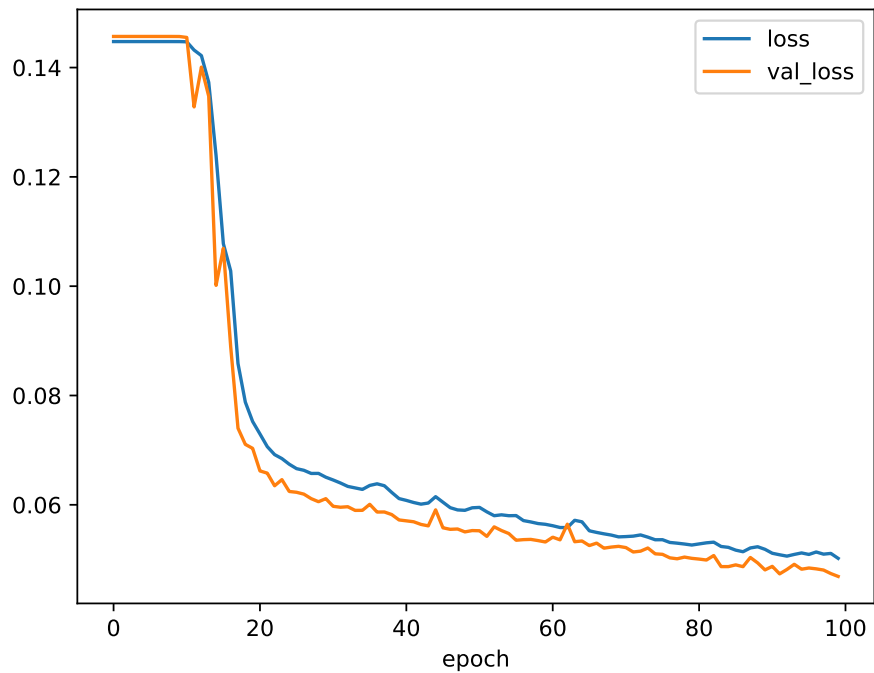


Figure F.5: Training history plot for feynman diagramm networks

In figure F.5 you see that, the training curve improves after the initial plateau first quite drastically, just to slow down later, and reach a validation loss below 0.05 at the end, which we are fairly happy with, since this means, that converted to booleans, only about 1 in 20 values is wrong<sup>184</sup>. More interestingly, you also see, that the validation loss is consistently lower than the training loss, which means, that even this network does not overfit, and we thus might be able to train a network on only  $O(1)$  events.

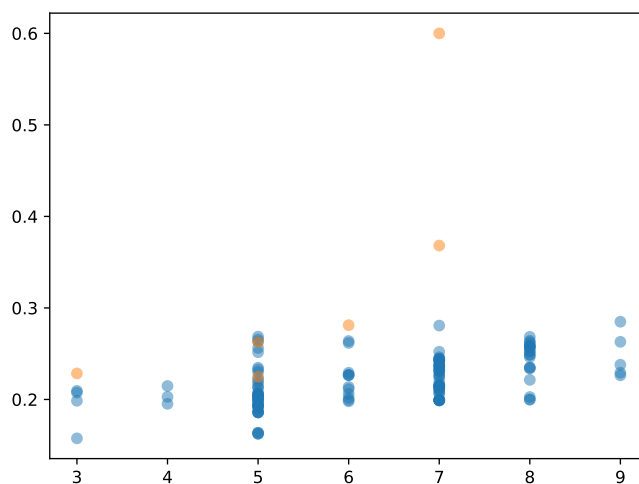


Figure F.6: Loss of each training graph compared to its number of lines.

One problem, that was prevalent for example in chapters 5 and F.1 is that complexity defines the loss at least as much as accuracy. This means, that when you have multiple levels

<sup>184</sup>Since the results are not booleans, this is only true for the average.

of complexity in your training data, you might not be able to differentiate more complex from more abnormal data points. So, since complexity for feynman graphs instinctively correlates to the number of lines, looking at the loss as a function of the number of lines is interesting (figure F.6): as you see, a bigger loss is generally more probable for higher line counts, but the relation is not so strong, as the difference is marginal. This suggests, that the complexity the network sees, is not the same, as the complexity we see, and it thus encodes more valuable information. Finally, to see this valuable information, we have to compare the loss to the loss of other diagrams. We first thought of looking at BSM diagrams, but different particles, which are fairly common in BSM graphs, would require different encoding, which is why we simply try 6 diagrams, which contribution vanishes. Those diagrams are shown in figures F.7 and F.8.

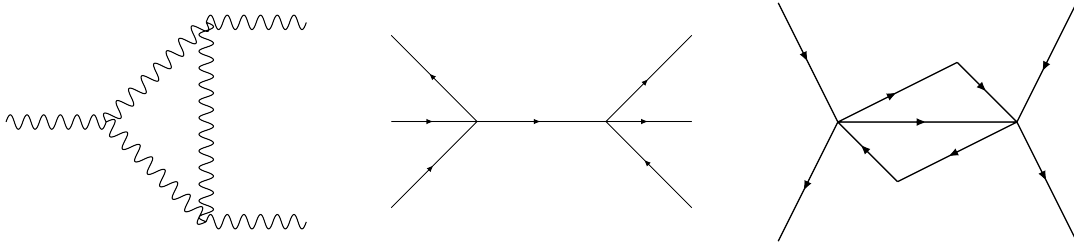


Figure F.7: vanishing diagrams (visualized using [8])

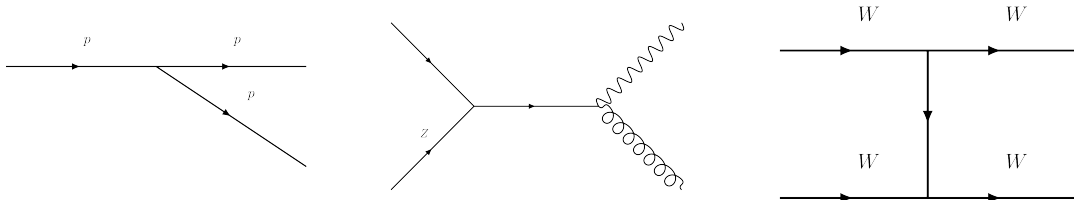


Figure F.8: more vanishing diagrams

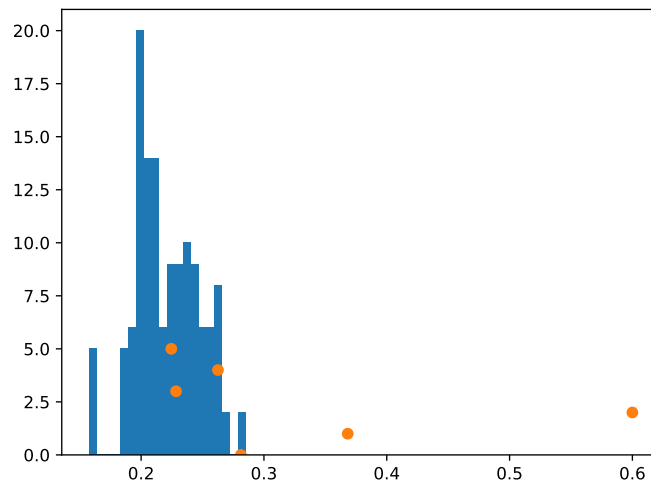


Figure F.9: Loss distribution for feynnp, with anomalies in orange showing their index by their height. We also cap the loss at 0.6, as the second anomaly would else result in a loss way over 10

If we plot the loss in figure F.9, for those diagrams into the loss distribution, you see some loss difference: the loss of those diagrams is generally bigger, and the lowest loss, that actually definitely overlaps the training loss, is achieved by the least complex diagram. This we can support by splitting the prediction by their line number. Those are the yellow dots in figure F.6. There you see, the difference between SM and non SM diagrams becomes even more clear.

### F.3.3 Whats next?

These six diagrams might suggest that this method works, but at the end, these are only 6 diagrams. So looking at more alternative diagrams is definitely a good idea. This might result in some diagrams that vanish, reaching fairly low losses, but might allow you to understand what the network considers complexity. In this note<sup>185</sup> You find the diagrams that have extreme losses for each line count. And here are two other things we noticed: Not every graph that is reconstructed actually exist. Through the original conversion, there are diagrams that could not be translated back to feynman diagrams

This might suggest, that weighting the adjacency matrix directly would be a good idea. you might also want to take a look at permutation invariant losses (see chapter 4.5). Secondly, most diagrams have two inputs, and the network is fairly good at reconstructing them

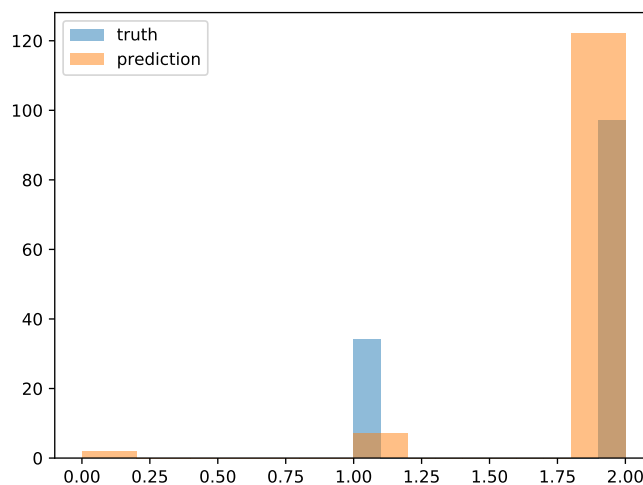


Figure F.10: input number histogramm for feynman networks

As you see, it might even be a bit too good, as it reconstructs even more 2 input diagrams. Checking for the errors in our representation might be useful. Finally, reproducibility and the applicability of oneoff networks might also be interesting here.

<sup>185</sup>From 42 diagramms with 5 notes LL601000002 has the lowest loss, while LL900000001 has the highest, for the 12 diagramms with 6 lines these are LL900000009 and LL900000005, for 44 of size 7 you find LL102000011 and LL210000001 and for the 21 graphs of size 8 the extremal cases are LL51100101 and LL522000005.

## F.4 Graph like generators and onoff initializers

Referenced in: [2.2] [C.5] [F.3.3]

As we had an example for autoencoder, and an example for a supervised graph network, we want to also highlight the possibility of having a graph as output. So we implement a graph generative adversarial network (A graph gan). We experimented with multiple different datasets, but finally use recipes.

The corresponding tutorial can be found at <https://grapa.readthedocs.io/en/latest/tipsy.html>

### F.4.1 Data generation

Finding recipes encoded as graphs is not really easy. So we use some simple text analysis to interpret recipes as graphs. Also, to make matters easier, we use drink recipes, since we don't need to implement the cooking steps. Each graph consists now out of at most 9 nodes of 45 one hot encoded ingredients. The list of ingredients is soft maxed, so that each ingredient is unique.

### F.4.2 Training

We generate 9 nodes, by decompressing 1 dimensional data twice by a factor of 3 into 9 nodes of 45 values, that we consider to be the coordinates of edges.

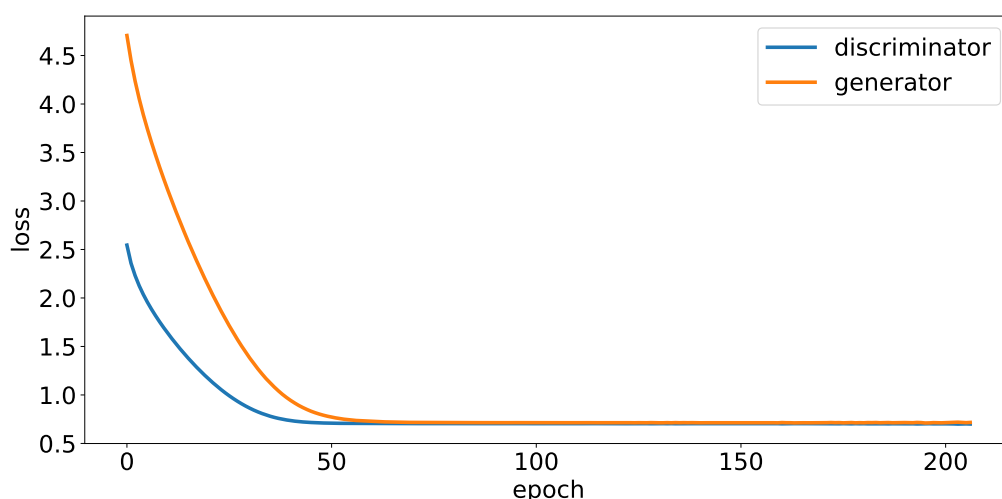


Figure F.11: Training curve for the generator. The training usually finishes when both losses are similar

Generative adversarial networks are usually stopped if both losses (see figure F.11 are similar. This point is also reached here without problems.

One thing we want to point out here, is an idea we had while trying to make this network work. You can use an oneoff network to initialize the discriminator. By removing the bias from the dense layers you train the discriminator like an oneoff network on the background data. And since this sets the mean of this dataset to one, but does not fix the mean of the alternative dataset, this can help the network converge. So here the training curve of a network, that trains for 10 epochs as an oneoff network first.

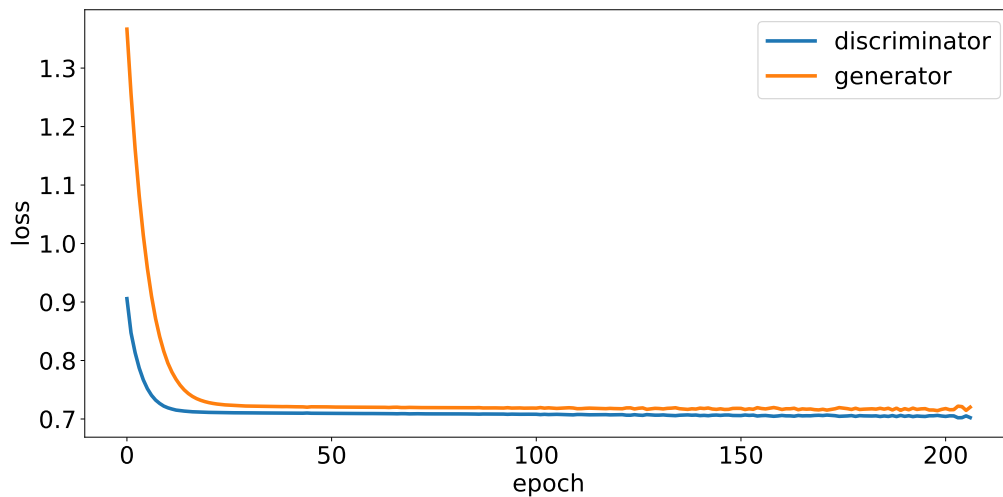


Figure F.12: Training curve now with an oneoff initializer (not shown) trained for 10 epochs

By comparing figure F.12 to F.11, you see that this training happens a lot quicker. So our oneoff initialization seems actually to a good idea

Also there is a website, at which you can take a look at example recipes <http://tipsy.pythonanywhere.com>

### F.4.3 Whats next?

For generating actual recipes you would need a way of encoding cooking steps.

Oneoff initialization should be tested on other datasets and by someone with more experience in training generative adversarial networks.

But the most interesting application might be other datasets. Here the obvious choice would be jets. This we tried, but the results are not good enough to show them here, as the distributions are nearly constant and require oneoff initializers to work at all. If you are interested in this, maybe just write me an email at [Simon.Kluettermann@gmx.de](mailto:Simon.Kluettermann@gmx.de).

## G Additional Figures

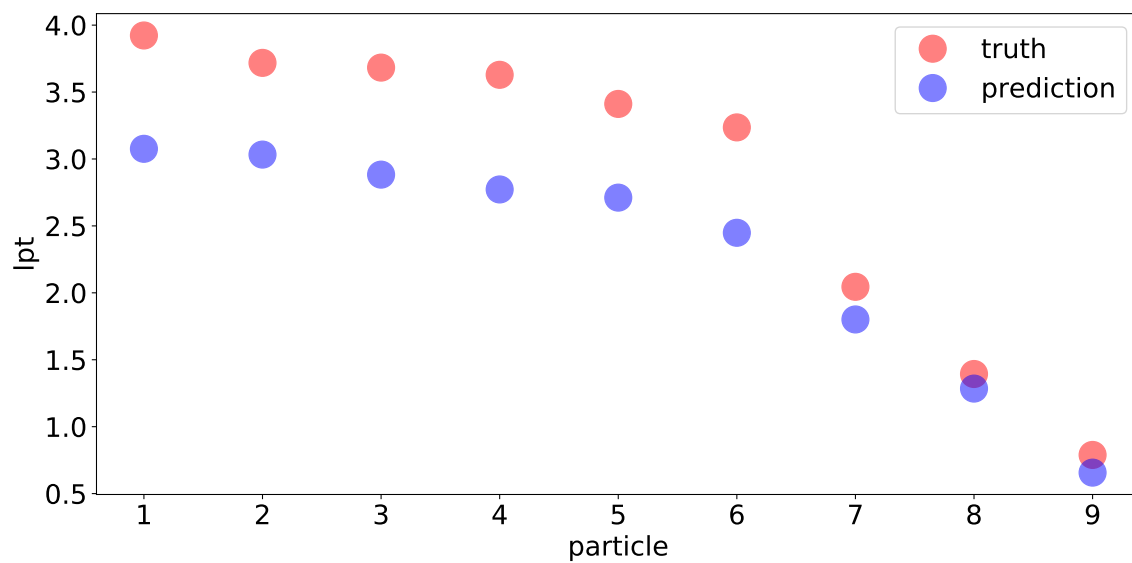


Figure G.1: Momentum reconstruction images for a 9 node image.

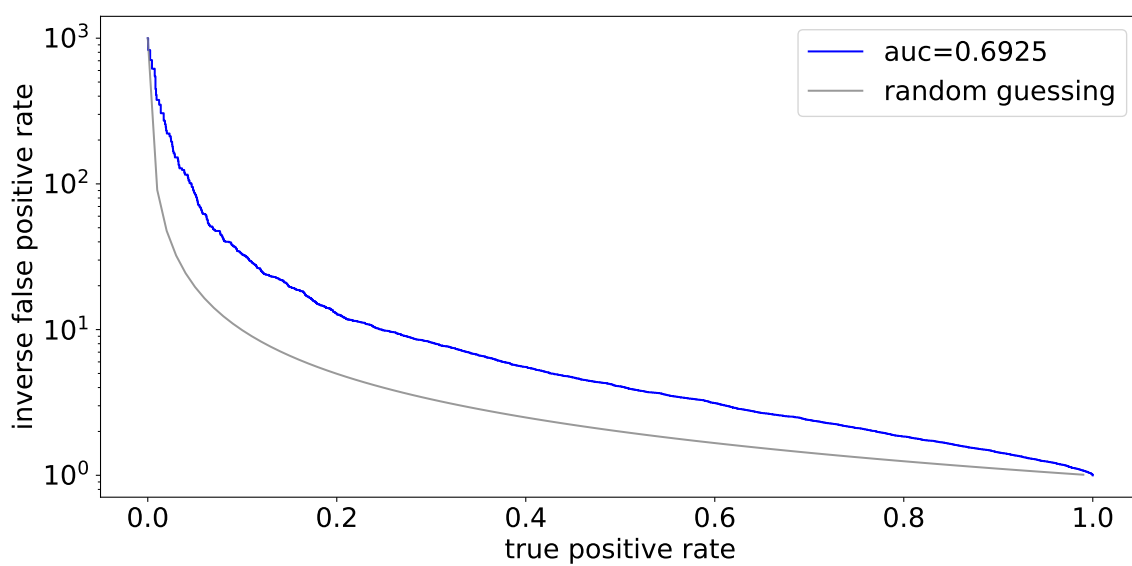


Figure G.2: ROC curve for a 16 node network trained on QCD

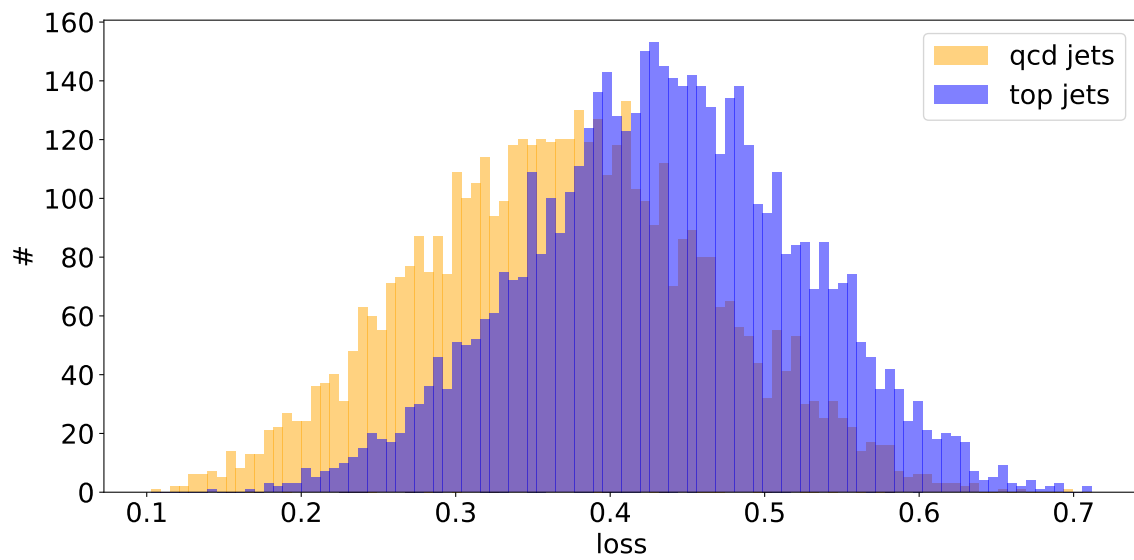


Figure G.3: Loss distribution for a 16 node network trained on QCD

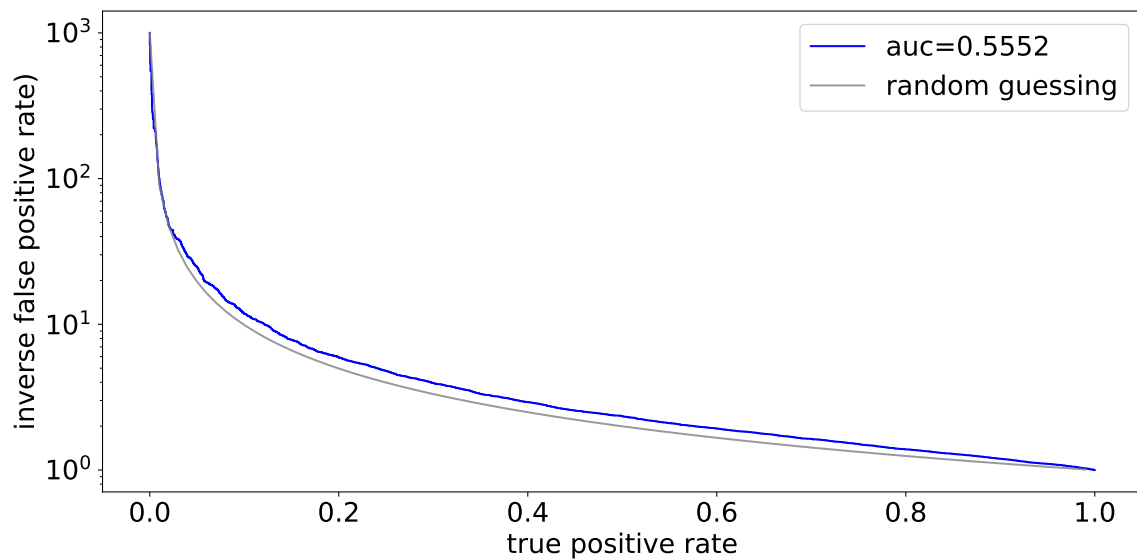


Figure G.4: Oneoff ROC curve for a 16 node network trained on QCD

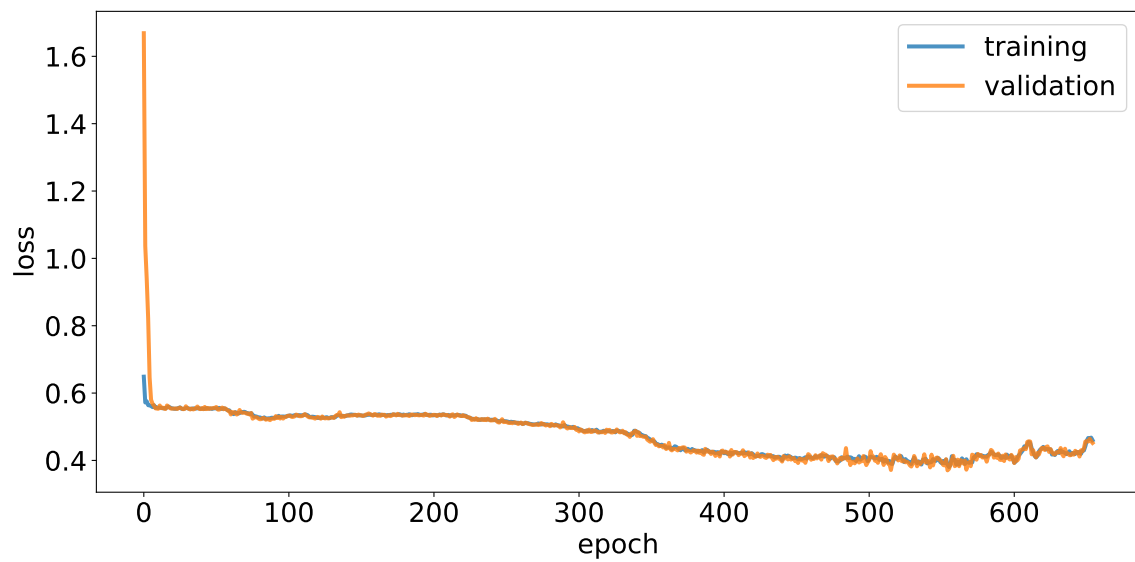


Figure G.5: Training history for a 16 node network trained on top

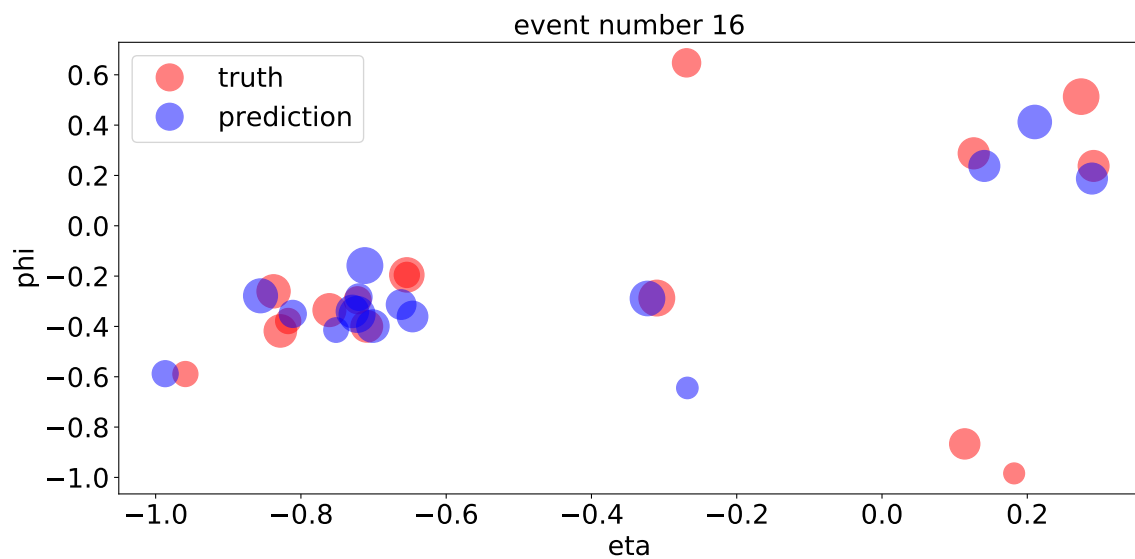


Figure G.6: Angular reconstruction for a 16 node network trained on top

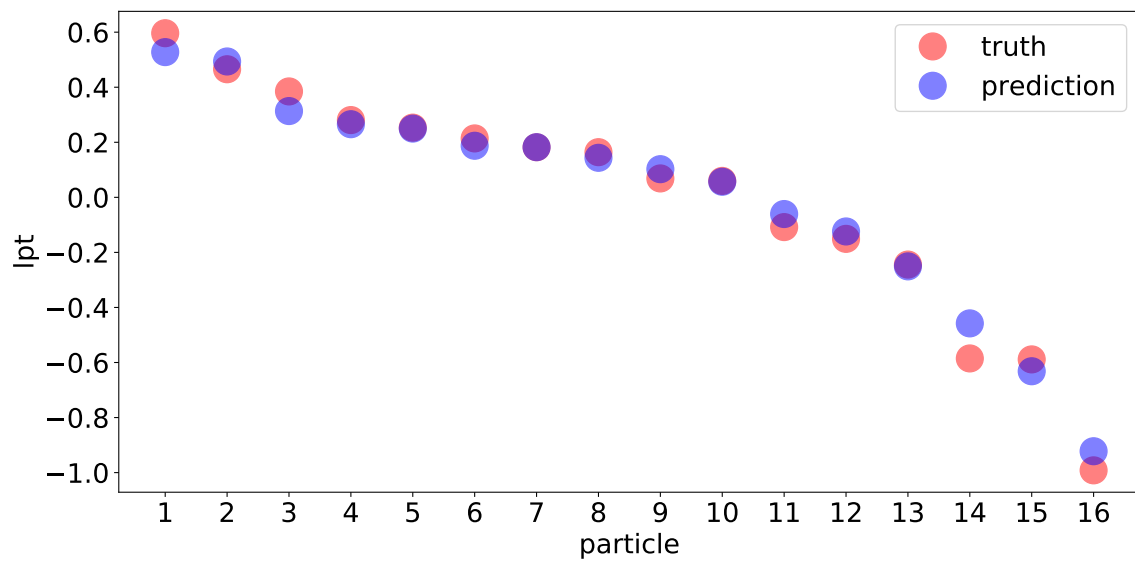


Figure G.7: Momentum reconstruction for a 16 node network trained on top

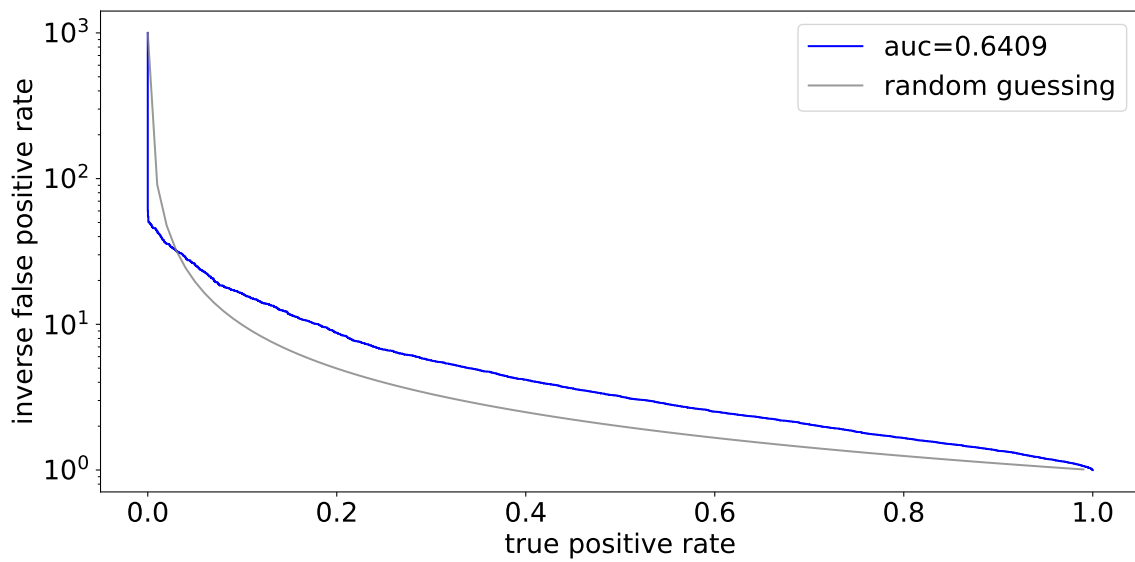


Figure G.8: ROC curve for a 16 node network trained on top

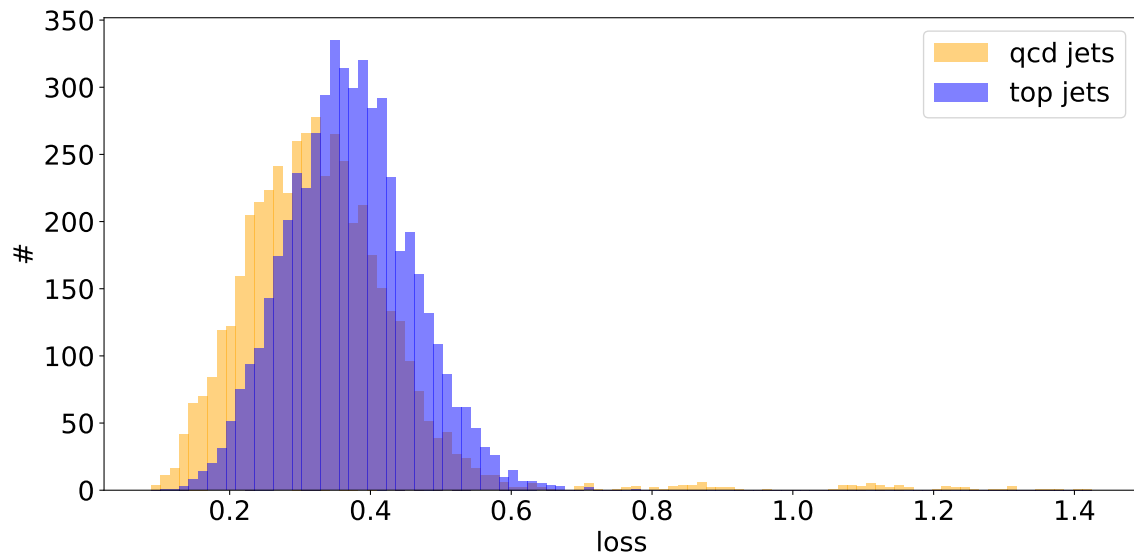


Figure G.9: Loss distribution for a 16 node network trained on top

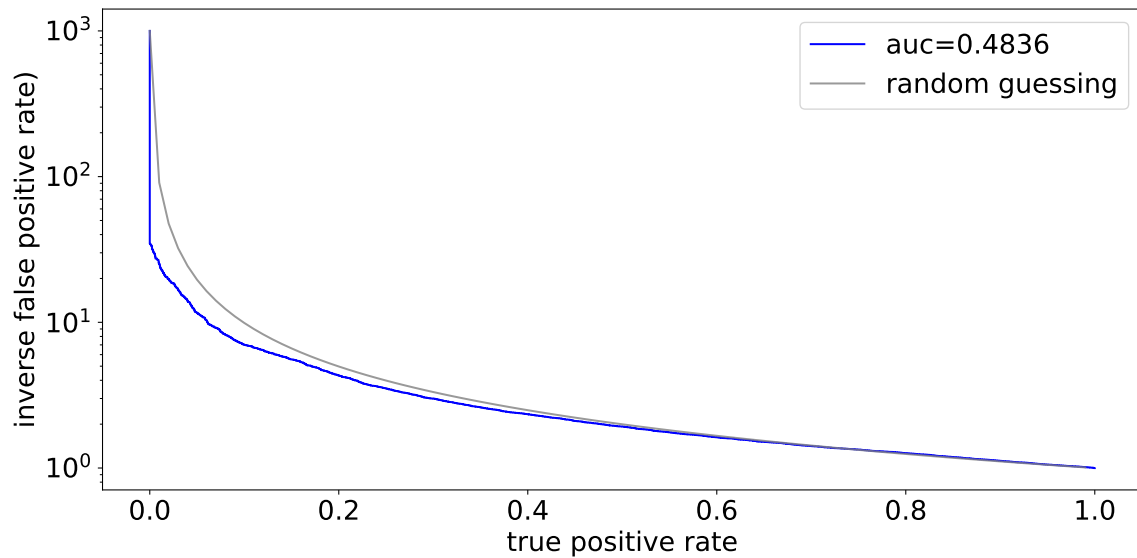


Figure G.10: Oneoff ROC curve for a 16 node network trained on top

## List of Figures

- |     |  |   |
|-----|--|---|
| 2.1 | A simple example on how an autoencoder can reduce the number of parameters that is needed to approximately encode an event. Instead of using two variables $x$ and $y$ to define each of the points, you can use only the $x$ value as latent space and an approximation of the $y$ value as a function of this latent space $x$ value . . . | 6 |
| 2.2 | Example images showing how an autoencoder can combine two images into one. Taken from [1], generated by Ember, Bruno and ArgonOl . . . . .   | 7 |
| 2.3 | A representation of the city regions as a graph: You can understand a map as a graph, where each region becomes a node and bridges between them represent edges. Here using a map from [2] . . . . .   | 9 |

3.1	A sample loss distribution, to explain how to calculate a ROC curve. To get your ROC curve, you have to choose every possible position of the parameter. Given one parameter value, everything with a loss higher than the parameter is classified as signal, and everything with lower loss as background. Your ROC curve is now the collection of all true and false positive rates for each possible parameter value . . . . .	11
3.2	A sample ROC curve plotted in way showing its AUC score . . . . .	12
3.3	A sample ROC curve showing the background rejection rate . . . . .	12
3.4	A sample reconstruction image . . . . .	15
3.5	A sample momentum reconstruction image . . . . .	15
3.6	A sample AUC Featuremap . . . . .	16
4.1	A $L_2$ reconstruction image, the reconstructed width is often lower than the input width. Here you see this best in $\phi$ . . . . .	21
4.2	Reconstruction image for a model that only remembers 3 nodes perfectly trained with an $L_1$ loss . . . . .	22
4.3	A $L_1$ reconstruction image, not working trivially, but also not working perfectly	23
4.4	Reconstruction image of an image like loss working well . . . . .	24
4.5	Training history for a 4 node network . . . . .	26
4.6	Angular reconstruction images for a 4 node image. . . . .	26
4.7	Momentum reconstruction images for a 4 node image. . . . .	27
4.8	Example training history for a 9 node network showing how NAN losses hurt the training procedure . . . . .	27
4.9	Angular reconstruction images for a 9 node image. . . . .	28
4.10	Loss distribution of our 4 particle network . . . . .	28
4.11	Roc curve for our 4 particle network . . . . .	29
4.12	AUC feature map for 4 nodes. . . . .	29
4.13	Loss distribution for the 9 particle network . . . . .	30
4.14	Roc curve for our 9 particle network . . . . .	30
4.15	Auc feature map for 9 nodes . . . . .	31
5.1	AUC score scaling through multiple batches of 4 nodes each . . . . .	33
5.2	Scaling autoencoder, by using dense networks instead of of graph ones . . . . .	33
5.3	An example of how we model AUC as a function of the overlapp of two gaussian peaks . . . . .	34
5.4	AUC as function of $c$ for two random gaussian double peaks that alone would reach AUCs represented by the horizontal lines . . . . .	35
5.5	AUC score as a function of the loss power (-3) for parts of a QCD jet, using 5 4 node networks combined in a way defined by the loss power . . . . .	36
5.6	AUC map for a simple network . . . . .	37
5.7	Average relative error by feature ( $\frac{\text{mean}( x-f(x) )}{\text{mean}( x )}$ ) . . . . .	38
5.8	2d histogram of angles comparing QCD vs top, here for example for the particle with the fourth highest transverse momentum . . . . .	38
5.9	Trivial width comparing angular scaling. Here we do not split each network in batches of 4 anymore, but simply calculate the AUC for each number of nodes up to 60. . . . .	39
5.10	Trivial width comparing angular scaling with $c$ addition. The reason for the falloff at the end might be the different shape in later indices of missing particles or the assumptions tested in appendix E.4 . . . . .	39
5.11	Roc curves for the invertibility of a 4 node model . . . . .	41
6.1	Aucmap for normally normalized networks, showing not much useful being learned. We train here on top jets to test the invertibility . . . . .	43

6.2	Invertible 4node network auc maps achieved by a normalization. Here trained on top jets . . . . .	45
6.3	Invertible 4node network auc maps achieved by a normalization. Here trained on qcd jets . . . . .	45
6.4	Double roc curve for the invertibility of a normalized networks . . . . .	46
6.5	More than 1500 Models, showing a clear relation between the network loss and the AUC score, each color represents slightly different training setups. It would be linear if the x-axis would be linear . . . . .	46
6.6	Invertibility for compressing 4 node networks into a 9 dimensional latent space. Notice the jump of the top AUC from above 0.5 to below 0.5 for lower losses. It means that when can use our loss to see if a trained network generates an invertible classifier. For compression sizes lower than 9 this jump does not appear	47
6.7	Reproducability comparison of for different training lengths . . . . .	47
6.8	AUC values for higher normalized batches by their training data . . . . .	48
6.9	AUC feature map for normalized top trained networks . . . . .	49
6.10	Distribution of the transverse momentum of the first particle . . . . .	49
6.11	AUC feature map for a well normated network . . . . .	50
7.1	AUC Feature map for an on top trained autoencoder, using a good normalization	51
7.2	AUC score as a function of the epoch, trained on QCD, here for a graph oneoff network. Graph oneoffs are not used anymore in the following, but since they show the same relation as a dense oneoff network much cleaner, we use this curve here. As you see, the relation shows a maximum before the training ends. . . . .	53
7.3	Oneoff loss distribution for a network trained on qcd jets . . . . .	54
7.4	Oneoff Roc curve for a network trained on qcd jets . . . . .	54
7.5	Angular reconstruction images for a normalized network trained on QCD . . . . .	55
7.6	Momentum reconstruction images for a normalized network trained on QCD . . . . .	55
7.7	Oneoff loss distribution for a network trained on top jets . . . . .	56
7.8	ROC curve for a network trained on top jets . . . . .	56
7.9	Angular reconstruction images for a normalized network trained on top jets . . . . .	57
7.10	Momentum reconstruction images for a normalized network trained on top jets . . . . .	57
7.11	Invertibility of batches in oneoff networks . . . . .	58
7.12	Training history for a 16 node network trained on QCD . . . . .	59
7.13	Angular reconstruction for a 16 node network trained on QCD . . . . .	59
7.14	Momentum reconstruction for a 16 node network trained on QCD . . . . .	60
8.1	Angular distribution of ldm jets . . . . .	62
8.2	Momentum distribution of ldm vs lQCD jets . . . . .	62
8.3	Number size distribution of ldm jets . . . . .	62
8.4	ldm jet invertibility . . . . .	64
8.5	Oneoff ROC curve for quark gluon. Here both curves should be above the line representing random guesses, and as you see, this is the case, even though the quark line is very close to randomly guessing. . . . .	65
8.6	Oneoff ROC curve for lepton data. Again both curves should be above the random guessing line . . . . .	66
8.7	Comparing each dataset to each other dataset, using oneoff networks. We use red points to mark values below 0.5 . . . . .	67
A.1	Metrik of a topK layer for a 4 momenta input . . . . .	72
A.2	Training history for the 4 momentum input. . . . .	73
A.3	Training curve using sorting . . . . .	74
A.4	Training curve using without sorting . . . . .	74
A.5	Training history with a batchnormalization layer . . . . .	76

A.6	Training history without a batchnormalization layer . . . . .	76
B.1	Rastering each compression size for a 4 node network (normalized) and showing each loss . . . . .	78
B.2	Rastering each compression size for a 4 node network (normalized) and showing each AUC value (without oneoff networks) . . . . .	78
B.3	Typical metrik of unnormalized networks . . . . .	80
B.4	Typical metrik of normalized networks . . . . .	81
B.5	Example of a set of nodes that cannot perfectly be connected using a topK algorithm (here $k = 1$ ). The problem here is, that 2 nodes have the same distance. We connect in this case both . . . . .	82
B.6	Example of a set of nodes that cannot perfectly be connected using a topK algorithm (here $k = 1$ . The problem here is, we want to connect the last node to a node that has no more open connections. We solve this by using asymmetric adjacency matrices . . . . .	82
B.7	A graph representing chairs. . . . .	83
B.8	Training curve trained on 50k top jets . . . . .	84
B.9	Training curve trained on 5k top jets . . . . .	84
B.10	Double ROC curve for different training sizes . . . . .	85
B.11	Double ROC curve on oneoff networks comparing training sizes. You should see no actual difference here . . . . .	85
C.1	Training history for first working autoencoder . . . . .	88
C.2	A reconstruction image for this model, we choose here one of the best reconstructed events . . . . .	89
C.3	AUC feature map for this model . . . . .	90
C.4	Training history for a better autoencoder . . . . .	91
C.5	Reconstruction image for the same event as in the previous chapter, on the left for the current model, and on the righth for the previous one . . . . .	92
C.6	AUC feature map for this model . . . . .	92
C.7	A sample 6 node graph splitted using our algorithm . . . . .	96
C.8	A sample 6 node graph splitted how we would like to split it . . . . .	97
D.1	A simple old AUC by epoch plot for a unnormalized network with thus focus on angular data . . . . .	101
D.2	AUC as function of the losses without normalization using a trivial decompressor.102	
D.3	Comparison of multiple network, with other decompressors. green represents graph like and orange parameter like decompression. Compare this to figure 6.5 in chapter 6.1 . . . . .	103
D.4	AUC by epoch for oneoff showing a growing relation, that at some point starts to fall again . . . . .	103
D.5	Partial network combinations, AUC as function of the graph size(gs) comparing a combination with equal weights to one with a loss power of 3 . . . . .	104
E.1	A simple example of a shape, that an SVM cannot differentiate . . . . .	106
E.2	Reconstruction image for the comparison algorithm . . . . .	108
E.3	Two different widths of a background peak, resulting in different overlappings to the signal peak . . . . .	110
E.4	The three kinds of simulated AUC by loss behaviours . . . . .	111
E.5	AUC as function of the loss for a oneoff network . . . . .	112
E.6	On the top: The 5 least 7 like 7th in the training set. On the bottom: The 5 most 7 like not 7th in the evaluation sample. Our favorite image is that in the lower right corner, as you can clearly see that it is a 9, while also allowing you to see how it can be interpreted as a 7. . . . .	114

E.7	Antiinvertibility for ldm data, left using the autoencoder loss, and on right using oneoff networks . . . . .	116
E.8	Number of jet particles for ldm jets . . . . .	116
F.1	Loss distribution for social networks . . . . .	119
F.2	Oneoff loss distribution on social networks . . . . .	120
F.3	Training curves on the left without compression step and on the right with one . . . . .	121
F.4	Example image of the conversion used for feynman diagramms, transforming the left diagramm into the right one . . . . .	123
F.5	Training history plot for feynman diagramm networks . . . . .	124
F.6	Loss of each training graph compared to its number of lines. . . . .	124
F.7	vanishing diagrams (visualized using [8]) . . . . .	125
F.8	more vanishing diagramms . . . . .	125
F.9	Loss distribution for feynnp, with anomalies in orange showing their index by their height. We also cap the loss at 0.6, as the second anomaly would else result in a loss way over 10 . . . . .	125
F.10	input number histogramm for feynman networks . . . . .	126
F.11	Training curve for the generator. The training usually finishes when both losses are similar . . . . .	127
F.12	Training curve now with an oneoff initializer (not shown) trained for 10 epochs . . . . .	128
G.1	Momentum reconstruction images for a 9 node image. . . . .	129
G.2	ROC curve for a 16 node network trained on QCD . . . . .	129
G.3	Loss distribution for a 16 node network trained on QCD . . . . .	130
G.4	Oneoff ROC curve for a 16 node network trained on QCD . . . . .	130
G.5	Training history for a 16 node network trained on top . . . . .	131
G.6	Angular reconstruction for a 16 node network trained on top . . . . .	131
G.7	Momentum reconstruction for a 16 node network trained on top . . . . .	132
G.8	ROC curve for a 16 node network trained on top . . . . .	132
G.9	Loss distribution for a 16 node network trained on top . . . . .	133
G.10	Oneoff ROC curve for a 16 node network trained on top . . . . .	133

## List of Tables

3.1	4 fractions for evaluating boolean decision problems. Events are truly column and classified as row . . . . .	11
8.1	Cross invertibility auc scores trained on row to be compared to column . . . . .	68
C.1	Quality differences for different encoder with a learnable handling of the feature vectors . . . . .	94
C.2	Quality differences for different encoder with a fixed function . . . . .	94
C.3	Quality differences for different graph like decoder . . . . .	95
C.4	Quality differences for different param like decoder . . . . .	95
C.5	Quality difference for either running learnable sub graph updates or not . . . . .	96
E.1	Distribution of types of loss vs AUC plots for random gaussian double peaks . . . . .	112
E.2	Learned metrik values of oneoff networks trained on muon events . . . . .	114
E.3	Learned metrik values for a diagonal metrik oneoff networks trained on muon events . . . . .	115

## References

- [1] URL: [https://www.ostagram.me/static\\_pages/lenta?last\\_days=1000&locale=en](https://www.ostagram.me/static_pages/lenta?last_days=1000&locale=en).

- [2] URL: <https://www.google.de/maps/@54.7134816,20.5119317,4270m/data=!3m1!1e3>.
- [3] Jetting into the dark side: a precision search for dark matter. URL: <https://atlas.cern/updates/physics-briefing/precision-search-dark-matter>.
- [4] List of feynman diagrams. URL: <https://www.physik.uzh.ch/~che/FeynDiag/Listing.php>.
- [5] G. Aad, T. Abajyan, B. Abbott, J. Abdallah, S. Abdel Khalek, A.A. Abdelalim, O. Abdinov, R. Aben, B. Abi, M. Abolins, and et al. Observation of a new particle in the search for the standard model higgs boson with the atlas detector at the lh. *Physics Letters B*, 716(1):1–29, Sep 2012. URL: <http://dx.doi.org/10.1016/j.physletb.2012.08.020>, doi:10.1016/j.physletb.2012.08.020.
- [6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.
- [7] Ajiboye Abdulraheem, J. Abdul-Hadi, Abimbola Akintola, and A.O. Ameen. Anomaly detection in dataset for improved model accuracy using dbscan clustering algorithm. *African Journal of Computing, ICT*, Vol 8. No. 1, 03 2015.
- [8] Alec Aivazis. URL: <https://feynman.aivazis.com/>.
- [9] Johan Alwall, Michel Herquet, Fabio Maltoni, Olivier Mattelaer, and Tim Stelzer. Mad-graph 5: going beyond. *Journal of High Energy Physics*, 2011(6), Jun 2011. URL: [http://dx.doi.org/10.1007/JHEP06\(2011\)128](http://dx.doi.org/10.1007/JHEP06(2011)128), doi:10.1007/jhep06(2011)128.
- [10] AndiLi99. image combiner. <https://github.com/AndiLi99/image-combiner>, 2018.
- [11] Elias Bernreuther, Thorben Finke, Felix Kahlhoefer, Michael Krämer, and Alexander Mück. Casting a graph net to catch dark showers, 2020. arXiv:2006.08639.
- [12] Filippo Maria Bianchi, Daniele Grattarola, and Cesare Alippi. Mincut pooling in graph neural networks, 2020. URL: <https://openreview.net/forum?id=BkxfshNYwB>.
- [13] Jason Brownlee. Data leakage in machine learning, Aug 2020. URL: <https://machinelearningmastery.com/data-leakage-machine-learning/>.
- [14] Claudio Campagnari and Melissa Franklin. The discovery of the top quark. *Rev. Mod. Phys.*, 69:137–212, Jan 1997. URL: <https://link.aps.org/doi/10.1103/RevModPhys.69.137>, doi:10.1103/RevModPhys.69.137.
- [15] Anwesha Chakraborty, Trina Dutta, Sushmita Mondal, and Asoke Nath. Application of graph theory in social media. *INTERNATIONAL JOURNAL OF COMPUTER SCIENCES AND ENGINEERING*, 6:722–729, 10 2018. doi:10.26438/ijcse/v6i10.722729.
- [16] Francois Chollet et al. Keras, 2015. URL: <https://github.com/fchollet/keras>.
- [17] Connor W. Coley, Wengong Jin, Luke Rogers, Timothy F. Jamison, Tommi S. Jaakkola, William H. Green, Regina Barzilay, and Klavs F. Jensen. A graph-convolutional neural network model for the prediction of chemical reactivity. *Chem. Sci.*, 10:370–377, 2019. URL: <http://dx.doi.org/10.1039/C8SC04228D>, doi:10.1039/C8SC04228D.

- [18] Jorge Dukelsky, G. Dussel, Jorge Hirsch, and P. Schuck. The pairing interaction in nuclei: comparison between exact and approximate treatments. 07 2002.
- [19] Mohammad Esmalifalak. A data mining approach for fault diagnosis: An application of anomaly detection algorithm. *Measurement*, 05 2014.
- [20] Thorben Finke. Deep learning for new physics searches at the lhc, 2020.
- [21] Hongyang Gao and Shuiwang Ji. Graph u-nets, 2019. [arXiv:1905.05178](#).
- [22] F. Gianotti. Physics at the LHC. *Phys. Rept.*, 403:379–399, 2004. doi:10.1016/j.physrep.2004.08.027.
- [23] Dan Guest, Kyle Cranmer, and Daniel Whiteson. Deep learning and its application to lhc physics. *Annual Review of Nuclear and Particle Science*, 68(1):161–181, Oct 2018. URL: <http://dx.doi.org/10.1146/annurev-nucl-101917-021019>, doi: 10.1146/annurev-nucl-101917-021019.
- [24] Theo Heimel, Gregor Kasieczka, Tilman Plehn, and Jennifer M Thompson. QCD or What? *SciPost Phys.*, 6:30, 2019. URL: <https://scipost.org/10.21468/SciPostPhys.6.3.030>, doi:10.21468/SciPostPhys.6.3.030.
- [25] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015. [arXiv:1502.03167](#).
- [26] U. S. R. Murty J. A. Bondy. Graph theory with applications.
- [27] G. Kasieczka, T. Plehn, A. Butter, D. Debnath, M. Fairbairn, W. Fedorko, C. Gay, L. Gouskos, P. T. Komiske, S. Leiß, A. Lister, S. Macaluso, E. Metodiev, L. Moore, B. Nachman, K. Nordstrom, J. Pearkes, H. Qu, Y. Rath, M. Riegler, D. Shih, J. M. Thompson, and S. Varma. The machine learning landscape of top taggers. *arXiv: High Energy Physics - Phenomenology*, 7:014, 2019.
- [28] Gregor Kasieczka, Tilman Plehn, Jennifer Thompson, and Michael Russel. Top quark tagging reference dataset, March 2019. doi:10.5281/zenodo.2603256.
- [29] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2014. [arXiv:1312.6114](#).
- [30] Thomas N. Kipf and Max Welling. Variational graph auto-encoders, 2016. [arXiv:1611.07308](#).
- [31] Boris Knyazev, Xiao Lin, Mohamed R. Amer, and Graham W. Taylor. Image classification with hierarchical multigraph networks, 2019. [arXiv:1907.09000](#).
- [32] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL: <http://yann.lecun.com/exdb/mnist/> [cited 2016-01-14 14:24:11].
- [33] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network, 2014. [arXiv:1312.4400](#).
- [34] Cheng-Yuan Liou, Wei-Chen Cheng, Jiun-Wei Liou, and Daw-Ran Liou. Autoencoder for words. *Neurocomputing*, 139:84–96, 2014.
- [35] Fei Tony Liu, Kai Ting, and Zhi-Hua Zhou. Isolation forest. pages 413 – 422, 01 2009. doi:10.1109/ICDM.2008.17.

- [36] Hui Luo, Ming xing Luo, Kai Wang, Tao Xu, and Guohuai Zhu. Quark jet versus gluon jet: fully-connected neural networks with high-level features, 2019. [arXiv:1712.03634](#).
- [37] Thomas McCauley. Events with two electrons from 2010. URL: <http://opendata.cern.ch/record/304>.
- [38] Thomas McCauley. Events with two muons from 2010. URL: <http://opendata.cern.ch/record/303>.
- [39] Erik Norlander and Alexandros Sopasakis. Latent space conditioning for improved classification and anomaly detection, 2019. [arXiv:1911.10599](#).
- [40] Sydney Otten, Sascha Caron, Wieske de Swart, Melissa van Beekveld, Luc Hendriks, Caspar van Leeuwen, Damian Podareanu, Roberto Ruiz de Austri, and Rob Verheyen. Event generation and statistical sampling for physics with deep generative models and a density information buffer, 2019. [arXiv:1901.00875](#).
- [41] S. Oryn, X. Rouby, and V. Lemaitre. Delphes, a framework for fast simulation of a generic collider experiment, 2010. [arXiv:0903.2225](#).
- [42] Thomas Petsche, Angelo Marcantonio, Christian Darken, Stephen Hanson, Gary Kuhn, and Iwan Santoso. A neural network autoassociator for induction motor failure prediction. 02 1996.
- [43] Adrian Alan Pol, Gianluca Cerminara, Cecile Germain, Maurizio Pierini, and Agrima Seth. Detector monitoring with artificial neural networks at the cms experiment at the cern large hadron collider, 2018. [arXiv:1808.00911](#).
- [44] Huilin Qu and Loukas Gouskos. Jet tagging via particle clouds. *Physical Review D*, 101(5), Mar 2020. URL: <http://dx.doi.org/10.1103/PhysRevD.101.056019>, doi:10.1103/physrevd.101.056019.
- [45] Mohammad Mahdi Rezapour Mashhadi. Anomaly detection using unsupervised methods: Credit card fraud case study. *International Journal of Advanced Computer Science and Applications*, 10, 01 2019. doi:10.14569/IJACSA.2019.0101101.
- [46] N. Rohani and Changiz Eslahchi. Drug-drug interaction predicting by neural network using integrated similarity. *Scientific Reports*, 9, 2019.
- [47] Lukas Ruff, Robert Vandermeulen, Nico Goernitz, Lucas Deecke, Shoaib Ahmed Siddiqui, Alexander Binder, Emmanuel Müller, and Marius Kloft. Deep one-class classification. volume 80 of *Proceedings of Machine Learning Research*, pages 4393–4402, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL: <http://proceedings.mlr.press/v80/ruff18a.html>.
- [48] Jonathan Shlomi, Peter Battaglia, and jean-roch vlimant. Graph neural networks in particle physics. *Machine Learning: Science and Technology*, Oct 2020. URL: <http://dx.doi.org/10.1088/2632-2153/abbf9a>, doi:10.1088/2632-2153/abbf9a.
- [49] Jenni Sidey-Gibbons and Chris Sidey-Gibbons. Machine learning in medicine: a practical introduction. *BMC Medical Research Methodology*, 19, 03 2019. doi:10.1186/s12874-019-0681-4.
- [50] Torbjörn Sjöstrand. The pythia event generator: Past, present and future. *Computer Physics Communications*, 246:106910, Jan 2020. URL: <http://dx.doi.org/10.1016/j.cpc.2019.106910>, doi:10.1016/j.cpc.2019.106910.

- 
- [51] David Tax. One-class classification; concept-learning in the absence of counter-examples. 01 2001.
  - [52] B. B. Thompson, R. J. Marks, J. J. Choi, M. A. El-Sharkawi, Ming-Yuh Huang, and C. Bunje. Implicit learning in autoencoder novelty assessment. In *Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN'02 (Cat. No.02CH37290)*, volume 3, pages 2878–2883 vol.3, 2002. doi:10.1109/IJCNN.2002.1007605.
  - [53] Eric Wulff. Deep autoencoders for compression in high energy physics, 2020. Student Paper.
  - [54] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling, 2019. arXiv:1806.08804.