UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Informatica

Elaborato finale

New physics detection in High Energy Physics experiments through unsupervised Deep Learning

Supervisore
Mauro Brunato

Co-Supervisore
Marco Cristoforetti

Laureando
Andrea Nardelli

Anno accademico 2015/2016

Ringraziamenti

First and foremost, thanks to Marco Cristoforetti, my tutor during my internship at FBK which resulted in this work, for the precious time spent discussing. Thanks to Mauro Brunato, my supervisor, for the same reasons.

Thanks to everyone at the MPBA office which over the years contributed to my well-being or, in the last period, to this thesis. In no particular order: Claudia, Cesare, Ernesto, Valerio, Bart, Franch, Setareh, Azra, Rachele, Stefano, Isotta, Lucia, Ylenia, Michele, Bizz, Roberto, Calogero, Bepi, Margherita, Andrea, Marco

Thanks to my family and all of my friends. A special thanks to: my m8, Valentina, Rachele.

Contents

Summary

The work described in this document was realized during an internship at the MPBA (Predictive Models for Biomedicine & Environment) research unit at the ICT center of Fondazione Bruno Kessler.

In 2014, a machine learning competition titled The Higgs Boson Machine Learning Challenge was held jointly by CERN, the Center for Data Science of Paris-Saclay, Research at Google and many other organizations. The objective of the competition was firstly the research of advanced machine learning models to improve classification tasks in experiments of high energy physics and secondly to promote synergy between the data science community and researchers in this field. High energy physics is the branch of physics which studies the fundamental interactions of elementary particles and amongst other important discoveries, it is responsible for the discovery of the Higgs boson which completes the Standard Model of physics and in fact it was the focus of the challenge which studied data regarding a particular decay of the boson. As mentioned above, machine learning is used widely in this field, e.g. on-line classification methods are used to sift through the data and discard uninteresting background events from signal events which could potentially describe new physics phenomena, while off-line methods are used to analyze the saved events.

This thesis picks up on the work done in the challenge and experiments with new techniques and models that have become more popular since 2014 for their outstanding results in other domains: neural networks and deep learning. This work utilizes *autoencoders*, neural networks that are used to learn new representations of data. These are effectively trained to reconstruct their inputs while minimizing an objective function, typically mean squared error. By imposing constraints on the model, such as the size of the hidden layers, it is possible to use them for the purpose of dimensionality reduction. In addition, this project is inspired from unsupervised anomaly detection techniques. Anomaly detection is the identification of outliers or data points which do not belong to the expected pattern of a dataset. Autoencoders can be used for this purpose, by classifying data points that are reconstructed with high reconstruction error as anomalies. In the context of high energy physics, anomalies are unexpected events when compared to theoretical predictions. Strictly related to autoencoders are generative models such as variational autoencoders or generative adversarial networks.

The data used in this work is an extended dataset of the original challenge. It describes properties such as the energy, type or momentum of decayed particles from proton-proton collisions detected by the ATLAS experiment at CERN. The information is then collected into a vector of 30 real valued features. These features are grouped in two subsets, for primitive and derived features. The former are raw properties of the particles, while the latter are computed from the derived features and were inserted in the dataset by high energy physicists in order to enrich the dataset and possibly improve machine learning tasks. In addition, the proportion of background and signal events in the dataset did not reflect the real distribution of experimental data (only approximately 0.2% of events are signal). However, in order to build a working model for classification, the dataset was rescaled to contain almost $\frac{1}{3}$ signal events but each sample was weighted by a coefficient (to be used when computing the metric) based on its probability. The metric used to evaluate the models, called *Approximate Median Significance*, was unusual in standard machine learning. In addition the signal and background classes overlapped by a large degree in the feature space.

The results achieved in this work are poor from a metric perspective, possibly due to the difficulty outlined above. However, the pipeline shows promising results which could be applied to other, different types of data in this field.

All the work done for this thesis is available on my public repository (`https://github.com/Illedran/DL-HEP`) on GitHub. The code is written in Python using the TensorFlow and Keras libraries. Detailed information about other libraries used and their versions is available in the `requirements.txt` file in the root directory of the repository, together with documentation in or-

der to reproduce my results. The models were trained on a GeForce GTX 1080 GPU.

1 Introduction

This thesis work consists of exploration of machine learning techniques, in particular deep learning, applied to data coming from particle physics (also called high energy physics, HEP). This branch of physics is concerned with studying the fundamental interactions of elementary particles. Machine learning is standard for the analysis of data from HEP experiments [8], in particular for classification purposes. In the last two years, possible applications of deep learning techniques have been applied in this domain [12, 7] in addition to new explorations in generative models [4].

In 2014, a competition titled The Higgs Boson Machine Learning Challenge (HiggsML) [1] was hosted on Kaggle, an online platform for "predictive modelling and analytics competitions". The goal of the challenge was to explore the potential of advanced classification methods in order to improve the statistical significance of experiments related to the discovery of the Higgs boson. The challenge attracted an unprecedented number of participants, paving the way for long-lasting collaborations between data scientists and high energy physicists.

1.1 Physics motivation

In high energy physics, experimental data is used to verify theory predictions. A very well known example of this is the discovery of the Higgs boson, whose existance was theorized fifty years before its actual discovery. After acquiring experimental data (called *events*), a classification model is then used to determine whether events of interest (called *signal*) occurred in addition to background events. The goal is to find a region in the feature space in which there is a significant amount of *signal* events compared to what known background processes can explain. Once the region is fixed, a statistical test is applied to determine the significance of this amount and check whether the hypothesis that the event sample contains only background events can be rejected. If the probability that the excess has been produced by background processes falls below a limit, the new particle is deemed to be discovered.

Nowadays, HEP scientists use machine learning for real-time discrimination of background events by multi-stage classifiers implemented via an ensemble of decision trees. Additionally, offline analysis is used to optimize the selection region of signal events.

1.2 Methodology and goals

This project is inspired from unsupervised anomaly detection techniques. Anomaly detection is the identification of outliers or data points which do not belong to the expected pattern of a dataset. This work utilizes *autoencoders*, neural networks that are used to learn new representations of data. These are effectively trained to reconstruct their inputs while minimizing an objective function, typically mean squared error. By imposing constraints on the model, such as the size of the hidden layers, it is possible to use them for the purpose of dimensionality reduction. Autoencoders can be thought of as composed of two different submodels:

- An *encoder* learns a non-linear transformation of data in a new *latent* representation.
- A *decoder* utilizes the transformed data to recreate the original input.

Samples that are reconstructed with significantly high reconstruction error are then defined as anomalies. In the context of HEP experiments, anomalies are unexpected events when compared to theoretical predictions. Hence, the application of anomaly detection techniques such as the ones in this thesis work are used to aid in the search of new physics. Despite not having optimal datasets for this type of analysis, the data from the Kaggle competition was used in order to implement anomaly detection algorithms in HEP.

In this project, the autoencoder is trained only on background data, hypothesizing a higher reconstruction for signal events which are then defined anomalies. Closely connected to autoencoders are *variational autoencoders*, which can take into account the variability of the distribution of the data and can be used as a generative model after being trained. The last model examined in this work are generative adversarial networks. They represent a different approach to generative models in which two different submodels, a generator and a discriminator, compete against each other. Generative models trained in this way can be used to recreate the data and potentially, provide additional information regarding the cause of the anomaly.

The goal of this work is exploring the possibility of improvement in the detection of particles in high energy physics by means of the models described above.

1.3 Learning prospects

It is important to be clear about the goal: by restricting this work to unsupervised learning, this project does not aim to have results comparable to the original Kaggle competition. However, the exploration of possible contributions to particle physics through machine learning, especially through deep learning models, is still ongoing. Compared to classical machine learning, deep learning presents several different advantages:

- Scalability: deep learning can be used with extremely large amounts of data, such as the data from HEP, thanks to GPU-assisted training. Several frameworks also offer the possibility to distribute the network on different GPUs.
- Representation learning: the intuition behind deep learning is that data can be represented in many different ways. For example, an image can be seen as a matrix of values, or in a more abstract way, as a set of shapes or edges. Each layer can learn more abstract features, which can be examined for additional insights of the data.
- Performance: deep learning has provided state-of-the-art results in several different fields, such as computer vision, NLP (natural language processing), and many others.

It is worth mentioning that in the HiggsML challenge, the contestant that ranked first utilized an ensemble of neural networks, proving the validity of these models.

2 Data

The ATLAS and the CMS experiment at the Large Hadron Collider (LHC) at CERN in Geneva claimed the discovery of the Higgs boson. Its importance is considerable because it is the final ingredient of the Standard Model of particle physics. Data from these experiments was used in the HiggsML challenge, and it is the same data used in this thesis work.

The Higgs boson can *decay* through several different processes. After the initial discovery in 2012, the study of all different modes of decay increases confidence in the validity of the theory and helps define the new particle. A decay into specific particles is called a *channel*. The challenge focuses on improving the analysis for one specific channel called *Higgs to tau tau* ($H \to \tau\tau$).

Since the problem is the discovery of a new phenomenon, labeled examples of actual signal events in the real data are not available. Events are simulated by a complex simulator that generates events following the Standard Model and a model of the detector, taking into account noise and possible artifacts. However, only the simulations and not the simulator itself was made available.

2.1 Data acquisition pipeline

The LHC accelerates bunches of protons on a circular trajectory in both directions. When these bunches cross in the ATLAS detector, some of the protons collide generating hundreds of millions proton-proton collisions per second. Part of the kinetic energy of the protons is converted into new particles which are detected by sensors, producing a sparse vector of a hundred thousand dimensions. From this raw data, the type (electron, photon, muon, etc.), energy and 3D momentum of each particle are estimated. The objects involved in the decays of interest in the challenge and this project are electrons, hadronic taus, jets and missing transverse energy. For every event, the attributes of each particle are collected and transformed into the features used in our model. Afterwards, it is possible to infer properties of the decayed parent particle, continuing the inference chain until the heaviest primary particles.

The amount of data generated by these experiments is staggering. The LHC accelerates protons every 50 nanoseconds, which then generate an amount of collisions with a Poisson expectation between 10 and 35 depending on the conditions of the accelerator. This means hundreds of millions (approximately from 2×10^8 to 7×10^8) collisions per second. Each crossing of the particles in the detectors is called an *event*. Online classification methods reduce the event rate to about 400, still producing about one billion events and three petabytes of raw data per year.

2.2 Dataset division

In the original Kaggle competition, two datasets were given: a training dataset with 250000 samples and a test dataset with 5500000 samples. Both datasets contained 30 features. In addition, the training one contained two additional columns for the label "b" for background or "s" for signal and the weight of the event. This last column is very important and has a physical meaning. Given $\mathcal{S}$ the set of signal events and $\mathcal{B}$ the set of background events, then:

$$\sum_{i \in \mathcal{S}} w_i = N_s \text{ and } \sum_{i \in \mathcal{B}} w_i = N_b$$

where w_i is the weight of the i-th sample. N_s and N_b are the expected number of signal and background events during the time interval of 2012. On average, the weight of a signal event is about 300 times lower than a background event. This means that the ratio of background and signal events in this dataset do not reflect data from real experiments, which produce very imbalanced signal and background classes (only approximately 0.2% of all events are signal events). In order to help competitors

in creating reliable models, the simulated data was enriched with additional signal data.

After the competition, the full dataset was released, divided in the following subsets:

- `t`: Corresponds to the original training dataset.
- `b`: Corresponds to a subset of 100000 samples of the Kaggle test set. It was used for the public leaderboard.
- `v`: Corresponds to the remaining 450000 samples of the Kaggle test set. It was used for the private leaderboard.
- `u`: Corresponds to an additional 18238 samples, unused in the Kaggle competition.

Each set was also provided with weights (normalized over the subset and normalized over the whole dataset) and labels, if they were missing.

2.3 Features

All the 30 features in the dataset (with the exception of one, detailed below) are real-valued. The variables were prefixed with `PRI` (for PRImitives) for raw values measured by the detector and `DER` (for DERived) for quantities computed from the primitive features. These were created by physicists in order to aid in the classification procedure. Out of all the features, some are especially important and worth describing:

- `DER_mass_MMC`: This feature represents the estimated mass of the Higgs boson candidate. Obtained from a complex calculation, this feature may be undefined when the event is too far from the expected topology of a signal event.
- `PRI_*_phi`: These features represent azimuthal angles in the $[-\pi, +\pi[$ range of the various particles. As can be seen from their histograms, these features almost follow a uniform distribution and were discarded by most models in the competition and in this work due to the possibility of overfitting.
- `PRI_jet_num`: This is the only integer feature in the dataset and it reflects the number of jets in this event (values of 0, 1, 2 or 3; possible larger values were capped at 3). Based on this feature, several other features could have missing data: for example, if the number of jets were 0, all features regarding jets would have been undefined.

Imputing

The last feature described made imputing missing values a complex problem. Several data points did not have features for the missing jets, and while imputing the mean or median value would not be a problem from a computational point of view (as a matter of the fact, the competitor that ranked first used mean imputation), imputing data that does not exist would be hard to justify from a physics perspective. For this reason, we decided to create 4 different models for each possible value of the `PRI_jet_num` feature.

The feature `DER_mass_MMC` could also have missing values. However, when present, it had a considerable impact on the classification process due to its nature. In table 2.1, some statistics are reported about the presence of the feature in the whole dataset (similar proportions apply to the `t` dataset).

Imputing was not used by all models. For example tree-based models, which achieved good results in the competition, did not require feature normalization or imputing.

2.4 Difficulties

The complexity of the original challenge can be explained by two factors:

- The classes overlap. Signal events and background events overlap in values for a vast majority of the features as can be seen in fig. 2.1.

DER_mass_MMC	signal count	background count	total count	signal ratio
NaN	9189	115413	124602	7.37%
$\mathbb{R}^+$	270371	423265	693636	38.98%

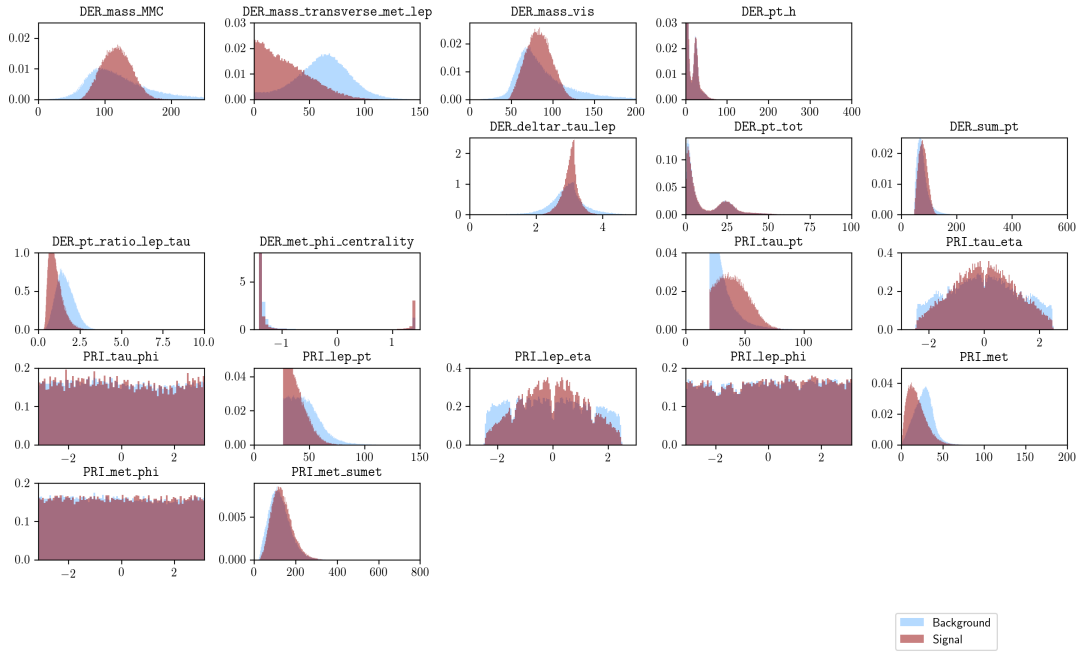
Table 2.1: Samples separated on the value for the DER_mass_MMC feature. Calculated on the complete dataset. The ratio of signal events that have an undefined value for DER_mass_MMC is much lower compared to the background. For this reason, some competitors decided to impute the minimum value for the column, or even 0 i.e. the estimated mass is zero as the Higgs boson is missing.

- The objective function, called *Approximate Median Significance* used in the competition is unusual. It is a function of the true positive and false positive rate of the model, weighted by the weight coefficient of each event. In perspective, the top ranking competitor achieved an AMS of 3.805, while the absolute maximum given a perfect classifier would have been 67.711. The AMS is defined as follows:

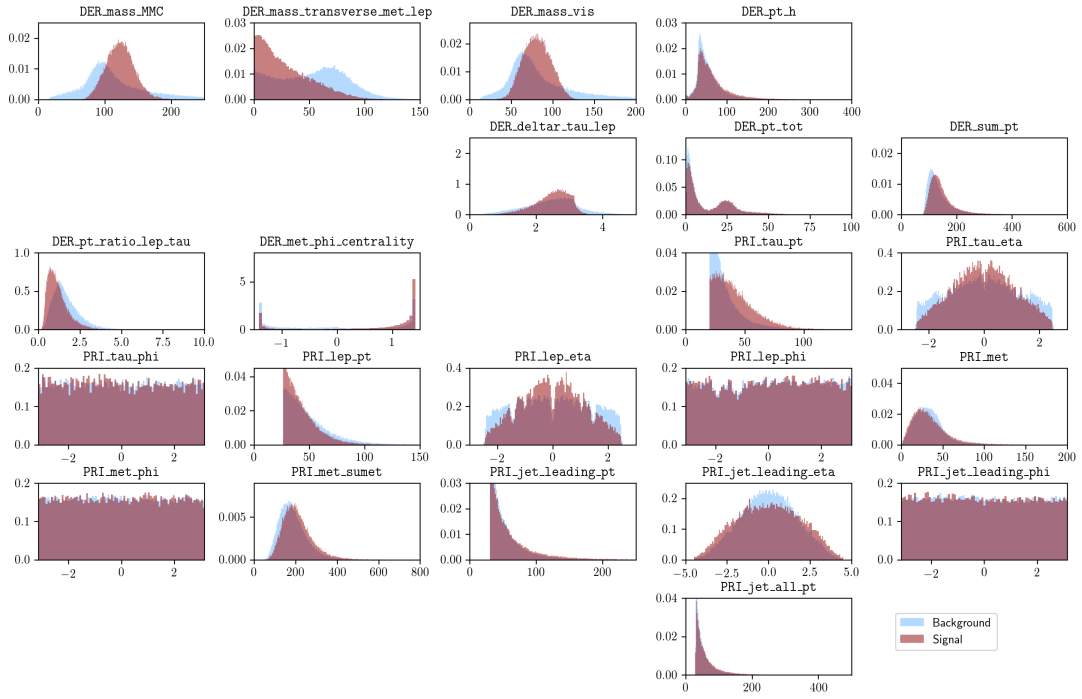
$$\text{AMS} = \sqrt{2\left((s+b+b_{reg})\ln\left(1+\frac{s}{b+b_{reg}}\right)-s\right)}$$

  Where s and b were respectively the sum of the weights for true positive events (signal classified as signal) and the sum of the weights for false positive events (background classified as signal). The term b_{reg} is a regularization term and was set to a constant $b_{reg} = 10$. In other words, the AMS is a metric that only takes in consideration the ability of the model to recognize *signal* events.

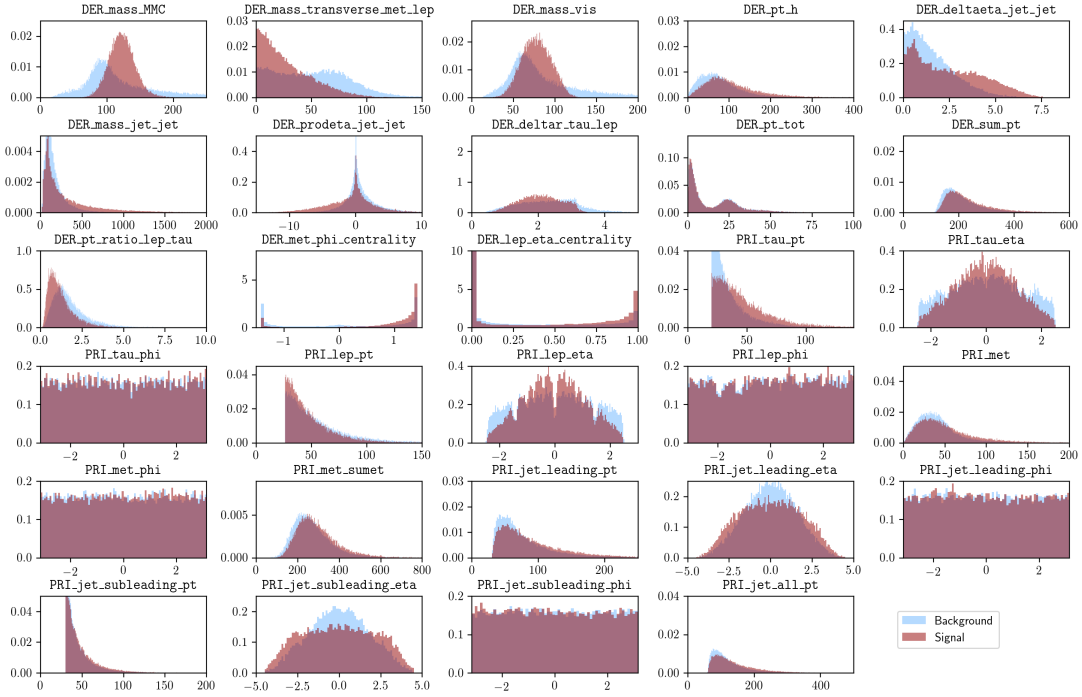
PRI_jet_num = 0



PRI_jet_num = 1



PRI_jet_num = 2



PRI_jet_num = 3

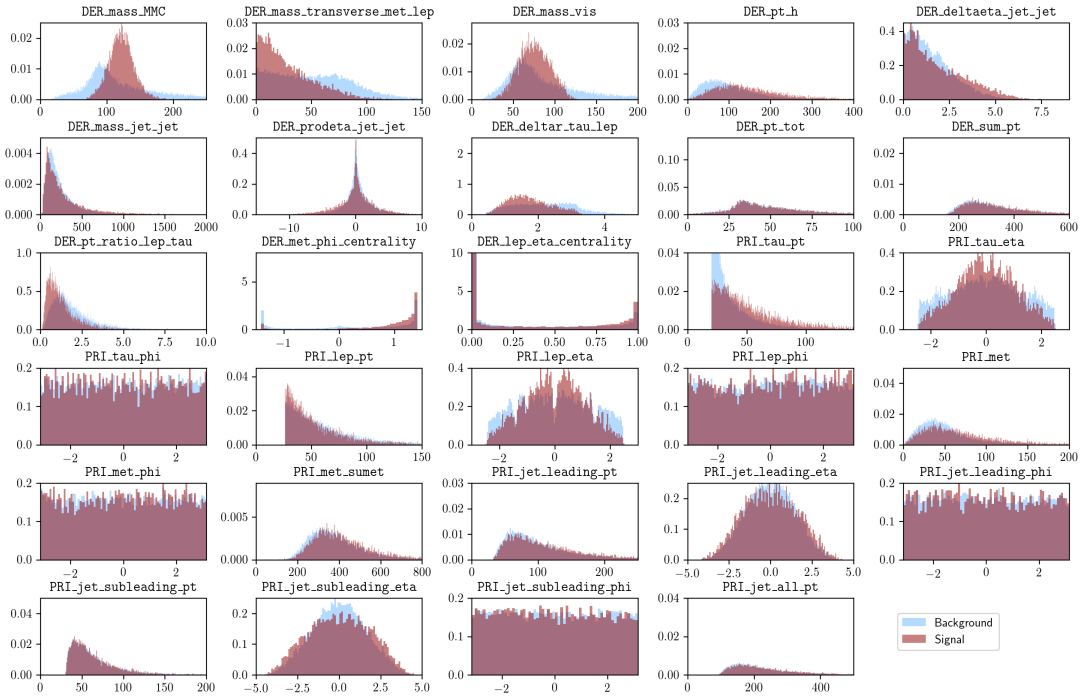


Figure 2.1: The four groups of plots represent histograms of each feature split on the possible values for the PRI_jet_num features. Plots may be missing when the features are not defined.

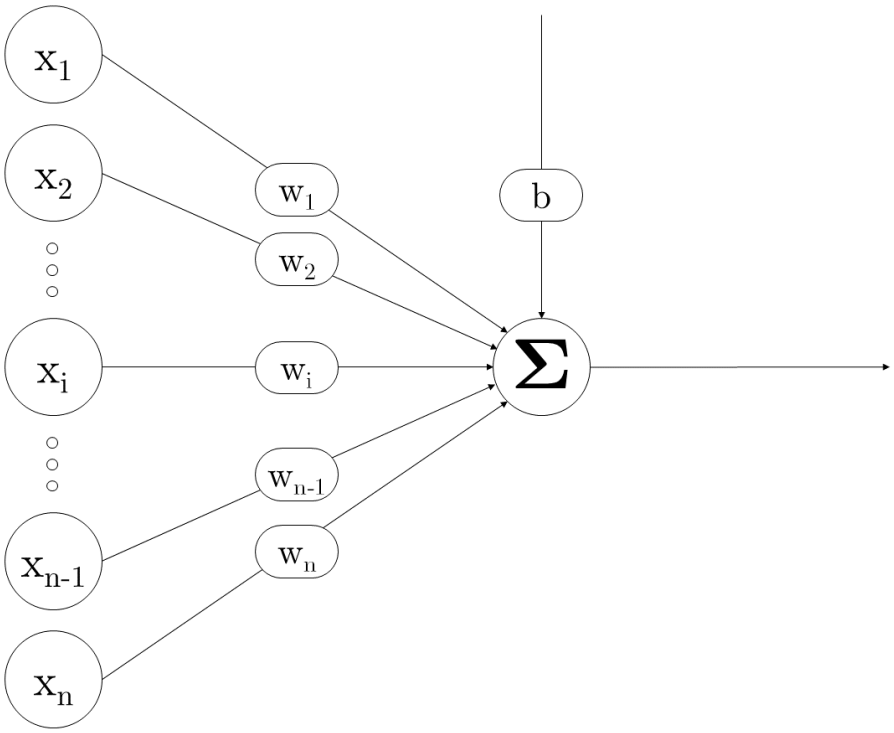


Figure 3.1: A perceptron with inputs $x_{1...n}$, weights $w_{1...n}$ and bias b.

3 Models

Artificial neural networks are computational systems loosely based on the way human neurons work. The simplest neural network is the perceptron (fig. 3.1), which applies just a simple transformation:

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b \tag{3.1}$$

Where:

- $\mathbf{x}$ is the input, a vector of real-valued data
- $\mathbf{w}$ is a real-valued vector of weights: here $\mathbf{w} \cdot \mathbf{x}$ symbolizes the dot product between $\mathbf{w}$ and $\mathbf{x}$,

$$\sum_{i=1}^{n} x_i w_i$$

- b is the bias of the perceptron, a scalar value independent from the input. This allows the perceptron to learn affine transformations.

The output value of the perceptron is then modified by an *activation function*. This mirrors the rate of action potential firing in the cell. In classical machine learning, the perceptron is used for linear

classification purposes and the activation function takes the form of the Heaviside step function:

$$g(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

Non-linear activation functions can be used as well. For example, the sigmoid function can be used for logistic regression in order to limit the output to be continuous in the $]0, 1[$ range:

$$g(x) = \frac{1}{1 + e^{-x}}$$

The perceptron can then be trained by gradient descent using the *delta rule*.

Application of delta rule for logistic regression

Let y be the binary target and $\hat{y}$ be the output value of $g(f(\mathbf{x}))$, the activation function applied to the output from the perceptron. The objective is the minimization of the error function,

$$E = \frac{1}{2}(y - \hat{y})^2 \tag{3.2}$$

The partial derivative of the error with respect to each weight w_i in the perceptron is

$$\frac{\partial E}{\partial w_i}$$

Replacing E with eq. (3.2):

$$\frac{\partial E}{\partial w_i} = \frac{\partial(\frac{1}{2}(y - \hat{y})^2)}{\partial w_i}$$

By applying the chain rule twice:

$$\begin{aligned} &\frac{\partial(\frac{1}{2}(y - \hat{y})^2)}{\partial w_i} = \\ =&\frac{\partial(\frac{1}{2}(y - \hat{y})^2)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_i} = \\ =&\frac{\partial(\frac{1}{2}(y - \hat{y})^2)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial f(\mathbf{x})} \frac{\partial f(\mathbf{x})}{\partial w_i} \end{aligned}$$

Reducing the first term with the power rule:

$$\begin{aligned} &\frac{\partial(\frac{1}{2}(y - \hat{y})^2)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial f(\mathbf{x})} \frac{\partial f(\mathbf{x})}{\partial w_i} = \\ =& -(y - \hat{y}) \frac{\partial \hat{y}}{\partial f(\mathbf{x})} \frac{\partial f(\mathbf{x})}{\partial w_i} \end{aligned} \tag{3.3}$$

Note that:

$$\frac{\partial \hat{y}}{\partial f(\mathbf{x})} = \frac{\partial g(f(\mathbf{x}))}{\partial f(\mathbf{x})} = g'(f(\mathbf{x})) \tag{3.4}$$

By replacing eq. (3.4) in eq. (3.3):

$$\begin{aligned} &-(y - \hat{y}) \frac{\partial \hat{y}}{\partial f(\mathbf{x})} \frac{\partial f(\mathbf{x})}{\partial w_i} = \\ =& -(y - \hat{y}) g'(f(\mathbf{x})) \frac{\partial f(\mathbf{x})}{\partial w_i} \end{aligned} \tag{3.5}$$

The derivative of the sigmoid function is:

$$g'(x) = g(x)(1 - g(x)) \tag{3.6}$$

By replacing eq. (3.6) in eq. (3.5):

$$\begin{aligned} &-(y-\hat{y})g'(f(\mathbf{x}))\frac{\partial f(\mathbf{x})}{\partial w_i} = \\ =&-(y-\hat{y})g(f(\mathbf{x}))(1-g(f(\mathbf{x}))\frac{\partial f(\mathbf{x})}{\partial w_i} = \\ =&-(y-\hat{y})\hat{y}(1-\hat{y})\frac{\partial f(\mathbf{x})}{\partial w_i} \end{aligned}$$

Rewriting $f(\mathbf{x})$ with eq. (3.1):

$$\begin{aligned} &-(y-\hat{y})\hat{y}(1-\hat{y})\frac{\partial f(\mathbf{x})}{\partial w_i} = \\ =&-(y-\hat{y})\hat{y}(1-\hat{y})\frac{\partial(\sum_{i=1}^{n} x_i w_i + b)}{\partial w_i} \end{aligned}$$

Considering the i-th weight, the only relevant term of the sum is $x_i w_i$:

$$\frac{\partial x_i w_i}{\partial w_i} = x_i$$

Giving us the final gradient for one sample:

$$\begin{aligned} &-(y-\hat{y})\hat{y}(1-\hat{y})\frac{\partial(\sum_{i=1}^{n} x_i w_i + b)}{\partial w_i} = \\ =&-(y-\hat{y})\hat{y}(1-\hat{y})x_i \end{aligned}$$

By setting an appropriate positive constant η (learning rate) and moving in the opposite direction of the gradient we arrive at our final equation:

$$\frac{\partial E}{\partial w_i} = \nabla w_i = \eta(y-\hat{y})\hat{y}(1-\hat{y})x_i$$

Instead of being trained over the whole dataset, neural networks are trained with minibatches. The gradient is then averaged across the minibatch and applied to the weights. The reason behind this is twofold:

1. Training the model with minibatches requires less memory than computing the gradient over the whole dataset.
2. Using batches of a large enough size (eg. 32 or 64 or higher) allows the CPU or GPU to use optimized and fast operations for vector or matrix multiplication.

The perceptron can also be generalized to have multiple outputs and it can be arranged in networks, in which the output of one can be connected to input of another perceptron. A feed-forward neural network is one such structure, in which the connections between perceptrons do not form any cycles. By stacking perceptrons and arranging them into layers, a multi-layer perceptron is obtained. Additionally, we define the inputs of each perceptron to be the outputs of all nodes in the previous layer: this is called a fully connected layer. In this case, each layer computes a more complex transformation:

$$f(\mathbf{x}) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Where:

- $\mathbf{x}$ is the input, a vector of real-valued data.
- $\mathbf{W}$ is a real-valued matrix of weights.
- b is a vector, which represents the bias of each perceptron.

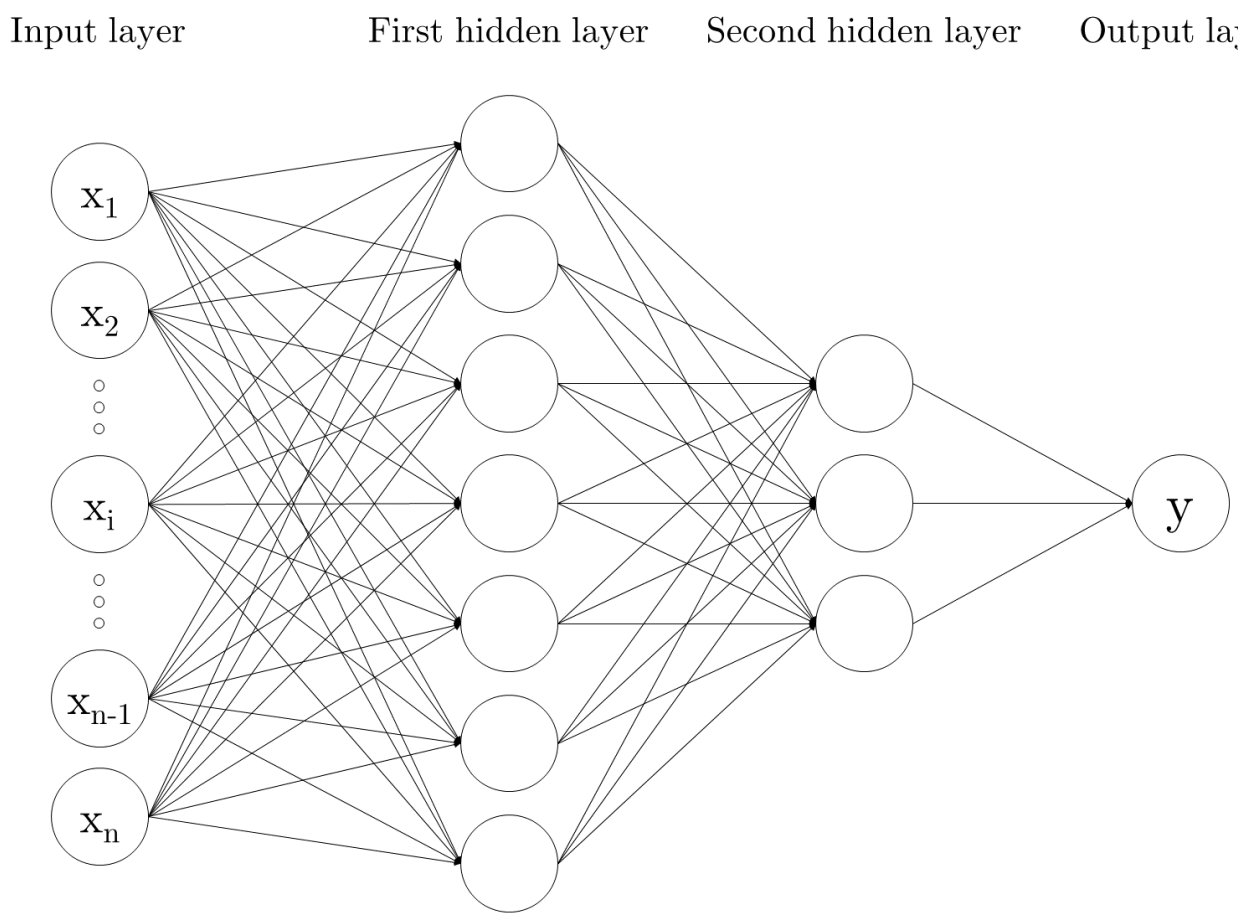


Figure 3.2: A multi-layer perceptron with two fully connected hidden layers.

- σ is the activation function of the layer (eg. sigmoid).

By stacking multiple layers, we finally arrive at a deep neural network (fig. 3.2). Layers different from the input or output layers are called *hidden* layers. The universal approximation theorem [9] states that any feed-forward neural network with a single hidden layer with a non-linear activation can approximate continuous functions in $\mathbb{R}^n$. This makes multi-layer perceptrons very versatile, as they can learn a wide variety of interesting functions. However, given this property and a large number of parameters, this type of models are very prone to overfitting.

Neural networks can be trained by backpropagation, a generalization of the delta rule made by applying the chain rule iteratively to each layer.

3.1 Autoencoder

Autoencoders (fig. 3.3) are neural networks used to learn reconstructions that are similar to their original input. By imposing a constraint on the number of neurons in the hidden layers, we expect the autoencoder to extract features that best represent the data in order to recreate it. Furthermore, we can stack additional layers and apply dimensionality reduction in a hierarchical manner, obtaining more abstract features in deeper layers. Autoencoders are separated in two parts, an encoder f and a decoder g. An encoder (here represented with two fully connected layers) maps input data to a latent representation $\mathbf{z}$:

$$\mathbf{z} = f(\mathbf{x}) = \sigma(\sigma(\mathbf{x}\mathbf{W}_{\mathbf{f_1}} + \mathbf{b}_{\mathbf{f_1}})\mathbf{W}_{\mathbf{f_2}} + \mathbf{b}_{\mathbf{f_2}})$$

Note that technically, each layer can have a different activation function. In this project, the activation functions corresponding to the latent layer is a simple linear layer, however other functions can be used to apply additional constraints to the latent representation.

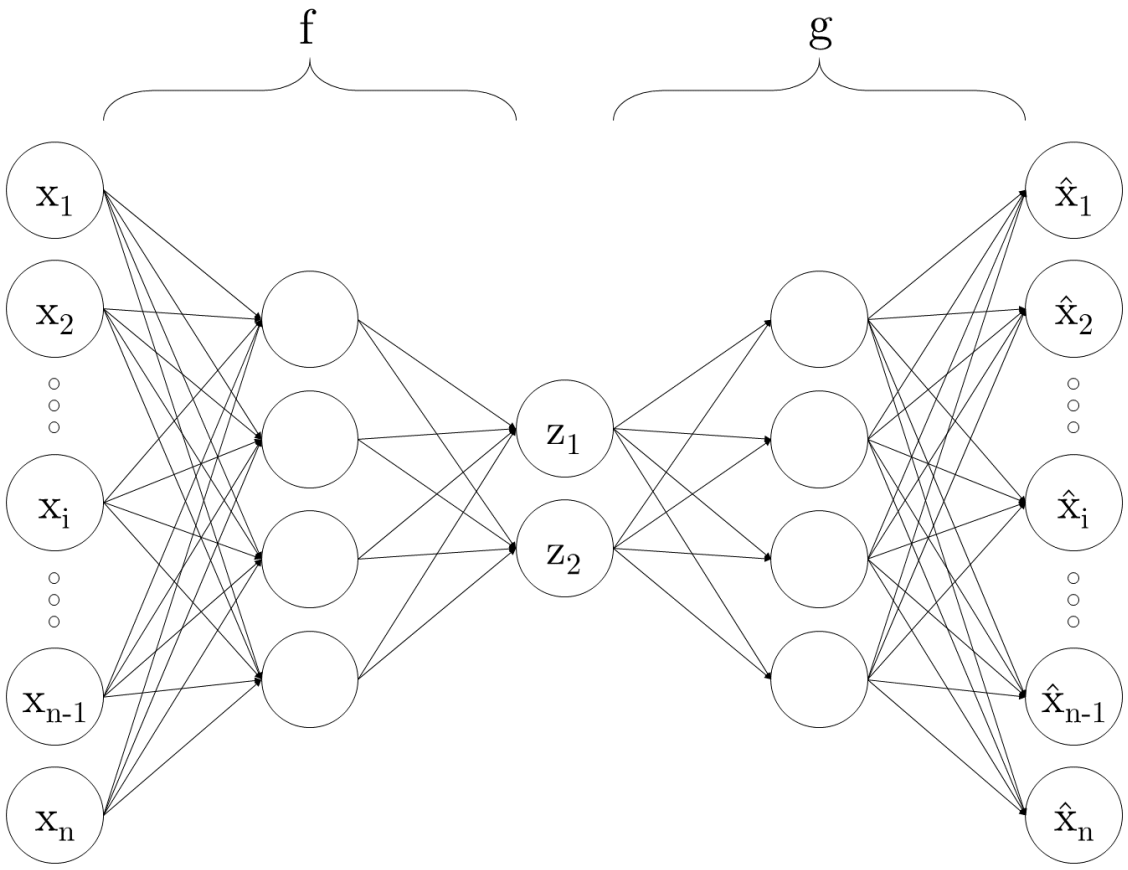


Figure 3.3: An autoencoder with a bi-dimensional latent representation. It is composed of two sub-models: encoder f and decoder g.

The decoder receives as input the latent representation $\mathbf{z}$ in order to recreate the original data. Clearly, due to the reduction in dimensionality of $\mathbf{z}$, the transformation applied by the decoder which results in the reconstructed input $\hat{\mathbf{x}}$ is lossy.

$$\hat{\mathbf{x}} = g(\mathbf{z}) = g(f(\mathbf{x})) = \sigma(\sigma(\mathbf{x}\mathbf{W}_{\mathbf{g_1}} + \mathbf{b}_{\mathbf{g_1}})\mathbf{W}_{\mathbf{g_2}} + \mathbf{b}_{\mathbf{g_2}})$$

Similarly to the encoder, the activation function corresponding to the output layer for the decoder is a linear layer. In the following pseudocode, the parameters of f and g will be respectively referred to as ϕ and θ. The algorithm for training an autoencoder is summarized in algorithm 1. Note that

Algorithm 1 Autoencoder training algorithm

1: **function** TRAINAE(dataset $\mathbf{X}$, encoder f, decoder g, batch size bs)
2: $\phi, \theta \leftarrow$ Initialize parameters
3: **repeat**
4: $\mathbf{X} \leftarrow$ SHUFFLE($\mathbf{X}$)
5: **for all** batch *in* ITERATEBATCHES($\mathbf{X}, bs$) **do**
6: **for all** $\mathbf{x}^{(i)}$ *in* batch **do** ▷ $\mathbf{x}^{(i)}$ denotes the i-th sample in the batch.
7: $\hat{\mathbf{x}}^{(i)} = g_\theta(f_\phi(\mathbf{x}^{(i)}))$
8: $E \leftarrow ||\mathbf{x}^{(i)} - \hat{\mathbf{x}}^{(\mathbf{i})}||$
9: $\nabla\phi^{(i)}, \nabla\theta^{(i)} \leftarrow$ Calculate gradients of E with backpropagation
10: $\phi, \theta \leftarrow$ Update parameters with average gradients over batch
11: **until** convergence of parameters.

operations that occur for each sample in a batch, i.e. lines 6 through 9 (included) are vectorized and executed with optimized code on CPU/GPU which offer considerable speedups. This applies to all operations applied to a batch of data. Line 2 mentions the initialization of the weights and bias of the neural network. The weights are initialized according to Xavier initialization [5]. The biases are initialized with 0. Line 4 shuffles the data as it has been observed that this accelerates the speed of convergence [2]. One pass over all the batches is called an *epoch*. The number of epochs elapsed can be used as a stopping condition for the repeat loop, however, depending on the data and the model, a different amount of passes over the data are required in order to converge.

Once the autoencoder is trained, it is possible to use it for anomaly detection purposes (algorithm 2) by setting a threshold t on the error. Autoencoders are a deterministic model. The model learns the

Algorithm 2 Anomaly detection algorithm for autoencoder

function DETECTANOMALYAE(training dataset $\mathbf{X}_1$, dataset with anomalies $\mathbf{X}_2$, encoder f, decoder g, batch size bs, threshold t)
TRAINAE($\mathbf{X}_1, f, g, bs$)
for all $\mathbf{x}^{(i)}$ *in* $\mathbf{X}_2$ **do**
$\hat{\mathbf{x}}^{(i)} = g_\theta(f_\phi(\mathbf{x}^{(i)}))$
$E^{(i)} \leftarrow ||\mathbf{x}^{(i)} - \hat{\mathbf{x}}^{(i)}||$
if $E^{(i)} > t$ **then**
classify $\mathbf{x}^{(i)}$ as anomaly
else
classify $\mathbf{x}^{(i)}$ as non-anomaly

most efficient coding in the space it is constrained into that can be used to reconstruct the data. However, they are not generative models as the only way to create latent vectors is encoding data. Since no assumptions are made on the latent space, sampling from it in order to generate new samples is not possible. For example, it is not possible to know whether the component-wise mean of two latent vectors $\mathbf{z}_1$ and $\mathbf{z}_2$ results in a meaningful sample. Variational autoencoders aim to solve this problem.

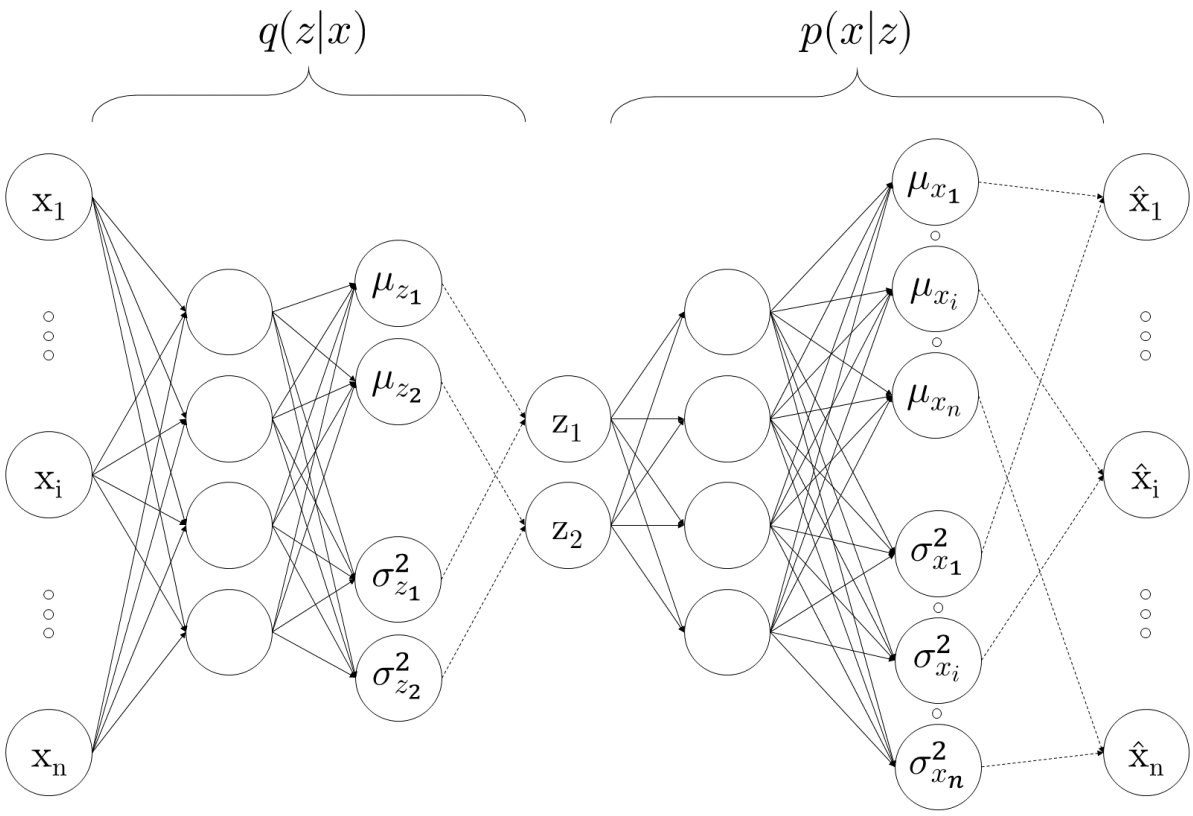


Figure 3.4: A variational autoencoder with a bi-dimensional latent representation. It is composed of two submodels: a probabilistic encoder q and probabilistic decoder p. Dotted lines represent sampling operations.

3.2 Variational Autoencoder

Variational autoencoders (fig. 3.4) resemble autoencoders in name and structure but the theory behind them originates with probabilistic graphical models. The idea is forcing the latent space to be stochastic for some type of distribution. The choice of latent distribution does not matter, as any distribution can be transformed into different distributions by a sufficiently complicated function. Hence, the latent distribution is usually chosen to be a multivariate gaussian with a diagonal covariance structure. The samples from this distribution will be denoted as z, and the dimensionality of distribution will be denoted as $|\mathbf{z}|$. We modify the encoder to output the parameters of the latent distribution, $\boldsymbol{\mu}_\mathbf{z}$ and $\boldsymbol{\sigma}^\mathbf{2}_\mathbf{z}$:

$$z|x \sim \mathcal{N}_{|\mathbf{z}|}(\boldsymbol{\mu}_\mathbf{z}, \boldsymbol{\Sigma}_z) \text{ with } \boldsymbol{\mu}_\mathbf{z} = [\mu_{z_1}, \ldots, \mu_{z_{|\mathbf{z}|}}] \text{ and } \boldsymbol{\Sigma}_z = \begin{bmatrix} \sigma^2_{z_1} & \ldots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & \sigma^2_{z_{|\mathbf{z}|}} \end{bmatrix}$$

The encoder modified in this way is called a *probabilistic encoder*, $q(z|x)$, generating parameters of the latent distribution which are dependent on ϕ, the weights and bias of the encoder. At this point we can modify the decoder in the same way, by having it generate parameters (which are dependent on θ, the weights and bias of the decoder) for the output distribution and calling it a *probabilistic decoder* $p(x|z)$. This is done in order to make explicit the dependence of x on z with the law of total probability:

$$p(x) = \sum_z p(x|z)p(z)$$

As we are reconstructing input data, i.e. doing a regression, we choose the output distribution $p(x|z)$ to be a multivariate gaussian distribution (for example, Bernoulli could be used for binary output).

The samples from this distribution will be denoted as x, and its dimensionality as $|\mathbf{x}|$.

$$x|z \sim \mathcal{N}_{|\mathbf{x}|}(\boldsymbol{\mu}_{\mathbf{x}}, \boldsymbol{\Sigma}_x) \text{ with } \boldsymbol{\mu}_{\mathbf{x}} = [\mu_{x_1}, \ldots, \mu_{x_{|\mathbf{x}|}}] \text{ and } \boldsymbol{\Sigma}_x = \begin{bmatrix} \sigma^2_{x_1} & \ldots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & \sigma^2_{x_{|\mathbf{x}|}} \end{bmatrix}$$

We can now think of the decoder as performing two different tasks:

1. The first few layers perform a transformation of the encoded latent variables into a different distribution that may be needed to best generate the data.

2. The remaining layers are used to generate parameters of the output distribution.

The objective function of a variational autoencoder is the maximization of the log likelihood for each sample:

$$\mathcal{L} = \log p_\theta(x)$$

Where $p_\theta(x)$ is a function of θ given the outcome x. In the following calculations, θ and ϕ are omitted. Rewriting the likelihood:

$$\begin{aligned} \mathcal{L} &= \log p(x) = \\ &= \sum_z q(z|x) \log p(x) = \\ &= \sum_z q(z|x) \log \left(\frac{p(z,x)}{p(z|x)} \right) = \\ &= \sum_z q(z|x) \log \left(\frac{p(z,x)}{q(z|x)} \frac{q(z|x)}{p(z|x)} \right) = \\ &= \sum_z q(z|x) \log \left(\frac{p(z,x)}{q(z|x)} \right) + \sum_z q(z|x) \log \left(\frac{q(z|x)}{p(z|x)} \right) \end{aligned} \tag{3.7}$$

In eq. (3.7), the second term is the Kullback-Leibler divergence from the true posterior distribution $p(z|\mathbf{x})$ to the approximate posterior distribution $q(z|\mathbf{x})$. This term is non negative, but due to the universal approximation theorem, the probabilistic encoder can eventually approximate it with $q(z|\mathbf{x})$. Thus, the first term is referred to as the variational *lower bound* on the likelihood of the i-th sample.

$$\mathcal{L} = \mathcal{L}^V + \mathcal{D}_{KL}(q(z|x)||p(z|x)) \geq \mathcal{L}^V$$

We can thus estimate $\mathcal{L}$ by calculating $\mathcal{L}^V$:

$$\begin{aligned} \mathcal{L}^V &= \sum_z q(z|x) \log \left(\frac{p(z,x)}{q(z|x)} \right) = \\ &= \sum_z q(z|x) \log \left(\frac{p(x|z)p(z)}{q(z|x)} \right) = \\ &= \sum_z q(z|x) \log \left(\frac{p(z)}{q(z|x)} \right) + \sum_z q(z|x) \log \left(p(x|z) \right) = \\ &= -\sum_z q(z|x) \log \left(\frac{q(z|x)}{p(z)} \right) + \sum_z q(z|x) \log \left(p(x|z) \right) = \\ &= -\mathcal{D}_{KL}(q(z|x)||p(z)) + \sum_z q(z|x) \log \left(p(x|z) \right) \end{aligned} \tag{3.8}$$

The first term of eq. (3.8) is the Kullback-Leibler divergence between our inferred distribution over the latent variables and the true distribution. Again, the true latent distribution does not matter, but we can set $p(z)$ to be a centered isotropic multivariate gaussian $\mathcal{N}(0, \boldsymbol{I})$. This term forces the encoder to

generate parameters close to the standard isotropic gaussian: in this way, after training we can draw samples from it and use them as input for the decoder to generate new data. The second term of this last equation is the expected log likelihood of the reconstruction done by the probabilistic decoder p, taken with respect to the distribution of the latent variables encoded by q. In order to evaluate this term, we can sample $\mathbf{z}$ L times:

$$\begin{aligned} &- \mathcal{D}_{KL}(q(z|x)||p(z)) + \sum_z q(z|x) \log\left(p(x|z)\right) \approx \\ \approx &- \mathcal{D}_{KL}(q(z|x)||p(z)) + \frac{1}{L}\sum_{l=1}^{L} \log\left(p(x|z_l)\right) \end{aligned} \tag{3.9}$$

Given our choice of distributions for the prior $p(z)$ and the posterior $q(z|x)$, we can integrate the first term of eq. (3.9) analytically. In particular, as our multivariate gaussians are diagonal, we can calculate the Kullback-Leibler divergence respectively for each independant gaussian, and compute the sum. Given:

$$\begin{aligned} a(x) =& \mathcal{N}(\mu, \sigma^2) \\ b(x) =& \mathcal{N}(0, 1) \end{aligned}$$

The Kullback-Leibler divergence from b to a is:

$$\begin{aligned} \mathcal{D}_{KL}(a(x)||b(x)) &= \int a(x) \log\left(\frac{a(x)}{b(x)}\right) dx = \\ &= -\int a(x) \log b(x) dx + \int a(x) \log a(x) dx \end{aligned} \tag{3.10}$$

Calculating the first term in eq. (3.10):

$$\begin{aligned} &- \int a(x) \log b(x) dx = \\ &= - \int a(x) \log\left(\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}\right) dx = \\ &= - \int a(x) \log e^{-\frac{x^2}{2}} dx - \int -\frac{1}{2}\log(2\pi) a(x) dx = \\ &= \frac{1}{2}\log(2\pi) - \int -\frac{x^2}{2} a(x) dx = \\ &= \frac{1}{2}\log(2\pi) + \frac{1}{2}\int x^2 a(x) dx \end{aligned} \tag{3.11}$$

The second term of eq. (3.11) is the second order moment of a gaussian distribution:

$$\int x^2 a(x) dx = \mu_a^2 + \sigma_a^2 \tag{3.12}$$

Plugging eq. (3.12) into eq. (3.11):

$$\begin{aligned} &\frac{1}{2}\log(2\pi) + \frac{1}{2}\int x^2 a(x) dx = \\ =&\frac{1}{2}\log(2\pi) + \frac{\mu_a^2 + \sigma_a^2}{2} \end{aligned} \tag{3.13}$$

The second term in eq. (3.10) is the negative *entropy* of a gaussian distribution:

$$-\int a(x) \log a(x) dx = \frac{1 + \log(2\sigma_a^2\pi)}{2} \tag{3.14}$$

Putting it together:

$$\begin{aligned} &-\int a(x)\log b(x)dx + \int a(x)\log a(x)dx = \\ =&\frac{1}{2}\log(2\pi) + \frac{\mu_a^2+\sigma_a^2}{2} - \frac{1+\log(2\sigma_a^2\pi)}{2} = \\ =&\frac{1}{2}\log(2\pi) + \frac{\mu_a^2+\sigma_a^2}{2} - \frac{1}{2} - \frac{1}{2}\log(2\pi) - \frac{\log\sigma_a^2}{2} = \\ =&\frac{1}{2}(\mu_a^2+\sigma_a^2-1-\log\sigma_a^2) \end{aligned} \tag{3.15}$$

Considering the j-th element of the two vectors of parameters, then:

$$\mu_a = \mu_{z_j} \text{ and } \sigma_a^2 = \sigma_{z_j}^2$$

Then, going back to the first term of eq. (3.9) and computing the sum over all the gaussians:

$$-\mathcal{D}_{KL}(q(z|x)||p(z)) = \frac{1}{2}\sum_{j=1}^{|\mathbf{z}|}(1+\log\sigma_{z_j}^2 - \mu_{z_j}^2 - \sigma_{z_j}^2) \tag{3.16}$$

This term is called the *latent loss* of the variational autoencoder.

In order to compute the second term of eq. (3.9), we must sample L times. However, this is a costly operation, especially when taking into account that these steps are repeated for each sample. The original paper presenting variational autoencoders states that "the number of examples L per datapoint can be set to 1, as long as the minibatch size was large enough, e.g. 100"[11]:

$$\frac{1}{L}\sum_{l=1}^{L}\log\left(p(x|z_l)\right) \approx \log\left(p(x|z)\right)$$

The maximum likelihood estimation depends on the choice of distribution for $p(x|z)$. As it was chosen to be a multivariate gaussian, we can compute the sum over the log likelihoods of each independent gaussian. The log likelihood function for a gaussian distribution for a sample x is defined as:

$$\log\mathcal{L}(\mu,\sigma^2) = -\frac{1}{2}\log(2\pi\sigma^2) - \frac{(x-\mu)^2}{2\sigma^2}$$

Calculating the sum over all the independent gaussian distributions:

$$\sum_{i=1}^{|\mathbf{x}|}\log\mathcal{L}(\mu_{x_i},\sigma_{x_i}^2) = -\sum_{i=1}^{|\mathbf{x}|}\frac{1}{2}\log(2\pi\sigma_{x_i}^2) + \frac{(x_i-\mu_{x_i})^2}{2\sigma_{x_i}^2}$$

This term mirrors the reconstruction loss of the regular autoencoder. As the backpropagation algorithm performs gradient descent, we want to move in the opposite direction of the variational lower bound, which means we want to minimize $-\mathcal{L}^V$. Putting it all together, the loss function for a sample x is:

$$-\mathcal{L}^V = \sum_{i=1}^{|x|}\frac{1}{2}\log(2\pi\sigma_{x_i}^2) + \frac{(x_i-\mu_{x_i})^2}{2\sigma_{x_i}^2} - \frac{1}{2}\sum_{j=1}^{|z|}(1+\log\sigma_{z_j}^2 - \mu_{z_j}^2 - \sigma_{z_j}^2) \tag{3.17}$$

The algorithm for training a variational autoencoder is summarized in algorithm 3. Note that compared to the regular autoencoder, the variational autoencoder does not reconstruct $\hat{\mathbf{x}}$ as part of its training process. Once the variational autoencoder is trained, it is possible to use it for anomaly detection purposes by setting a threshold t on the error (algorithm 4).

Algorithm 3 Variational autoencoder training algorithm

function TRAINVAE(dataset $\mathbf{X}$, prob. encoder q, prob. decoder p, batchSize bs)
  $\phi, \theta \leftarrow$ Initialize parameters
  repeat
    $\mathbf{X} \leftarrow$ SHUFFLE($\mathbf{X}$)
    for all batch *in* ITERATEBATCHES($\mathbf{X}, bs$) **do**
      for all $\mathbf{x}^{(i)}$ *in* batch **do**
        $\boldsymbol{\mu}_{\mathbf{z}}^{(i)}, \boldsymbol{\sigma}_{\mathbf{z}}^{2^{(i)}} = q_\phi(\mathbf{x}^{(i)}))$
        $\boldsymbol{\Sigma}^{(i)} \leftarrow$ Create diagonal matrix from vector $\boldsymbol{\sigma}_{\mathbf{z}}^{2^{(i)}}$
        $\mathbf{z}^{(i)} \sim \mathcal{N}_{|z|}(\mathbf{z}_\mu^{(i)}, \boldsymbol{\Sigma}^{(i)})$ ▷ Draw a sample $\mathbf{z}$ from the latent distribution.
        $\boldsymbol{\mu}_{\mathbf{x}}^{(i)}, \boldsymbol{\sigma}_{\mathbf{x}}^{2^{(i)}} = p_\theta(\mathbf{z}^{(i)}))$
        $E^{(i)} \leftarrow$ Calculate error using eq. (3.17)
        $\nabla\phi^{(i)}, \nabla\theta^{(i)} \leftarrow$ Calculate gradients of E with backpropagation
      $\phi, \theta \leftarrow$ Update parameters with average gradients over batch
  until convergence of parameters.

Algorithm 4 Anomaly detection algorithm for variational autoencoder

function DETECTANOMALYVAE(training dataset $\mathbf{X}_1$, dataset with anomalies $\mathbf{X}_2$, prob. encoder q, prob. decoder p, batch size bs, threshold t)
  TRAINVAE($\mathbf{X}_1, q, p, bs$)
  for all $\mathbf{x}^{(i)}$ *in* $\mathbf{X}_2$ **do**
    $\boldsymbol{\mu}_{\mathbf{z}}^{(i)}, \boldsymbol{\sigma}_{\mathbf{z}}^{2^{(i)}} = q_(\mathbf{x}^{(i)}))$
    $\boldsymbol{\Sigma}^{(i)} \leftarrow$ Create diagonal matrix from vector $\boldsymbol{\sigma}_{\mathbf{z}}^{2^{(i)}}$
    $\mathbf{z}^{(i)} \sim \mathcal{N}_{|z|}(\mathbf{z}_\mu^{(i)}, \boldsymbol{\Sigma}^{(i)})$
    $\boldsymbol{\mu}_{\mathbf{x}}^{(i)}, \boldsymbol{\sigma}_{\mathbf{x}}^{2^{(i)}} = p_(\mathbf{z}^{(i)}))$
    $E^{(i)} \leftarrow$ Calculate error using eq. (3.17)
    if $E^{(i)} > t$ **then**
      classify $\mathbf{x}^{(i)}$ as anomaly
    else
      classify $\mathbf{x}^{(i)}$ as non-anomaly

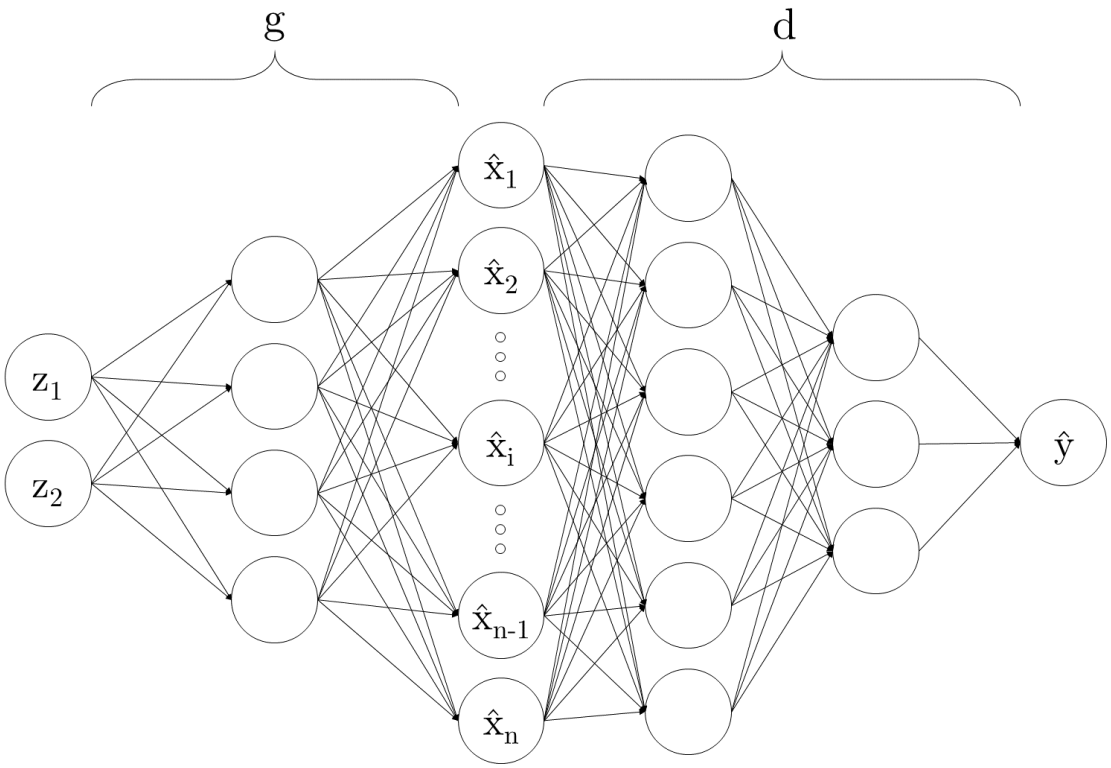


Figure 3.5: A generative adversarial network is composed of two different submodels: a generator $g(\mathbf{z})$ and a discriminator $d(\mathbf{x})$.

3.3 Generative Adversarial Network

A generative adversarial network (fig. 3.5), first introduced in [6], represent a different view on generative models. The model is composed of two different submodels competing against each other. In particular, the first model g, the generator, is tasked with mapping data from a latent representation to a desired distribution. The second model d, the discriminator, is tasked with classifying whether the input data belongs to the real dataset or is generated by g and output a probability value. In other words, the objective of the generator is to maximize the error of the discriminator, which corresponds to maximizing the entropy of the output. Given that we are performing a binary classification, the entropy for a single sample is:

$$H(y) = -y \log y - (1-y) \log(1-y)$$

We want to maximize this value:

$$\frac{\partial H(y)}{\partial y} = 0 \tag{3.18}$$

Solving for y:

$$\begin{aligned} \frac{\partial H(y)}{\partial y} =& \frac{\partial (y \log y - (1-y) \log(1-y))}{\partial y} = \\ =& -(\log y + 1) - (-\log(1-y) - 1) = \\ =& -\log y - 1 + \log(1-y) + 1 = \\ =& \log(1-y) - \log(y) \end{aligned} \tag{3.19}$$

Rewriting eq. (3.18) with eq. (3.19):

$$\begin{aligned} \log(1-y) - \log(y) &= 0 \\ 1 - 2y &= 0 \\ y &= \frac{1}{2} \end{aligned} \tag{3.20}$$

Intuitively, a probability value of $\frac{1}{2}$ means that the discriminator can not tell apart generated samples from data coming from samples from the true dataset. In order to push the discriminator towards this value, the generator must learn to generate data similar to the true dataset. On the other hand, the discriminator must be trained at the same time with generated samples (label 0) and samples from the dataset (label 1). The training procedure for the generative adversarial network is divided in two steps:

1. In the first step, the discriminator is trained. A mini batch of random latent variables $\mathbf{z}$ is sampled and used as input for the generator. Afterwards, a mini batch of data is obtained from the dataset, $\mathbf{x}$. The discriminator is trained to minimize the *cross entropy* for both classification tasks (i.e. by classifying generated samples as 0 and true data points as 1). The cross entropy for a sample is defined as:

   $$H(y, \hat{y}) = -y \log \hat{y} - (1-y) \log(1-\hat{y})$$

   The discriminator must be optimized both in respect to generated samples ($y = 0$ and $\hat{y} = d(g(\mathbf{z}))$),

   $$-\log(1 - g(d(\mathbf{z})))$$

   and true data ($y = 1$ and $\hat{y} = d(\mathbf{x})$):

   $$-\log d(\mathbf{x})$$

   Putting it together, the loss for the discriminator is:

   $$L_d = -\log d(\mathbf{x}) - \log(1 - g(d(\mathbf{z}))) \tag{3.21}$$

2. In the second step, the generator is trained (the parameters of the discriminator are kept fixed). An alternative way to look at the objective for the generator is to generate values which, once used as input for the discriminator, generate values close to 1 ($y = 1$ and $\hat{y} = d(g(\mathbf{z}))$):

   $$L_g = -\log d(g(\mathbf{z})) \tag{3.22}$$

The training procedure for a generative adversarial is summarized in algorithm 5. Note that the update of the parameters for both models can be delayed until after each training iteration. Anomalies can then be classified as samples which the discriminator can classify easily into the generated or true classes. For example given a threshold $t = 0.1$, data points for which the discriminator outputs values in the $(0, 0.1)$ or $(0.9, 1)$ range can be classified as anomalies. The algorithm is summarized in algorithm 6.

Algorithm 5 Generative adversarial network training algorithm

function TrainGAN(dataset $\mathbf{X}$, generator g, discriminator d, batchSize bs)
  $\phi, \theta \leftarrow$ Initialize parameters
  repeat
    $\mathbf{z} \leftarrow$ LatentSampling(bs)
    $\mathbf{x} \leftarrow$ SampleData($\mathbf{X}, bs$)
    $\mathbf{L_d} \leftarrow$ Calculate error using eq. (3.21) for each sample in the batch
    $\nabla\phi \leftarrow$ Calculate average gradient of $\mathbf{L_d}$ with backpropagation
    $\phi \leftarrow$ Update parameters of the discriminator

    $\mathbf{z} \leftarrow$ LatentSampling(bs)
    $\mathbf{L_g} \leftarrow$ Calculate error using eq. (3.22) for each sample in the batch
    $\nabla\theta \leftarrow$ Calculate average gradient of $\mathbf{L_g}$ with backpropagation
    $\theta \leftarrow$ Update parameters of the generator
  until convergence of parameters.

Algorithm 6 Anomaly detection algorithm for generative adversarial networks

function DetectAnomalyGAN(training dataset $\mathbf{X}_1$, dataset with anomalies $\mathbf{X}_2$, generator g, discrminator d, batch size bs, threshold t)
  TrainGAN($\mathbf{X}_1, g, d, bs$)
  for all $\mathbf{x}^{(i)}$ *in* $\mathbf{X}_2$ **do**
    $E^{(i)} \leftarrow d(\mathbf{x}^{(i)})$
    if $E^{(i)} > (1-t)$ or $E^{(i)} < t$ **then**
      classify $\mathbf{x}^{(i)}$ as anomaly
    else
      classify $\mathbf{x}^{(i)}$ as non-anomaly

4 Results

As outlined in section 2.3, the data was split (see table 4.1) into subsets according to their value for the `PRI_jet_num` feature in order to avoid the imputing problem. All the results outlined in this chapter have been obtained by training four different models on each split, calculating errors on samples in the corresponding data subset, then aggregating results and computing metrics.

4.1 Preprocessing

Features were scaled with standardization (unit mean and zero variance). The mean and variance of training data was used to standardize test data. In addition, the `phi` (angular) features in each subset were dropped, as suggested in the original competition due to their potential to overfit the model.

PRI_jet_num	signal count	background count	total count	signal ratio
0	82853	244518	327371	25.31%
1	90202	162680	252882	35.67%
2	84221	80806	165027	51.03%
3	22284	50674	72958	30.54%

Table 4.1: Samples separated on the value of `PRI_jet_num` feature. Calculated on the complete dataset.

Optimizers	Adam	SGD	Adadelta	RMSProp
Activation Functions	Relu	Sigmoid	Softplus	Tanh
Learning Rate	1×10^{-4}	1×10^{-5}	1×10^{-3}	1×10^{-2}
Batch Size	512	256	128	1024
Hidden Layers	[10, 5]	[25, 10, 5]	[100, 50, 25]	[500, 250, 100, 50]
Latent Dimensions	2	1	4	8

Table 4.2: Table of parameters and hyperparameters tested on the models. The first column represents the chosen value.

The `PRI_jet_all_pt` feature, corresponding to the sum of the transverse momentum of all the jets was dropped from the subset `PRI_jet_num`= 0 as from samples with zero jets as it had a fixed value of 0.

4.2 Model selection

The first step was determining good parameters for the models. Tunable parameters include the learning rate of the network, the size and number of layers, the initialization of biases and weights, the type of activation function used between layers, the dimensionality of the latent representation, the batch size, the type of optimizer used. An optimizer is an algorithm that tries to overcome some of the limitations of classic gradient descent, for example by adding momentum in the calculation of the gradient. Momentum reduces oscillations by adding a fraction of the gradient for the last update step to the current gradient. A brief overview of all (hyper)parameters tested is available in table 4.2. The activation function used in hidden layers is a rectified linear unit:

$$f(x) = \text{MAX}(0, x)$$

The output of the rectifier was also capped in order to prevent exploding gradients during backpropagation. For the same reason, weights and biases were initialized with standard values as described in section 3.1. The optimizer used is Adam [10], which obtains faster convergence and more robust training compared to standard gradient descent. The parameter that had the most impact on the training process and training time was the number and dimensions of the hidden layers. The plots in fig. 4.1 show how the loss function changes for an autoencoder for the different configurations examined. As can be seen, despite the loss function being lower with more and larger layers, its value fluctuates more, signifying an unstable training process. In addition, the difference in loss between signal and background events decreased accordingly i.e. more layers did not help the model discriminate between signal and background and could even overfit the data. The latent dimensions were chosen to be bi-dimensional in order to plot the latent space. For variational autoencoders, the same parameters as autoencoders were used. Generative adversarial networks were tested with a multitude of parameters. However, no combination was found which converged during the training procedure. The difficulty in achieving stable training for generative adversarial networks is well known, and it is possible that they can not be applied well to this kind of task, which differs from their original purpose e.g. computer vision. The networks were trained with dropout layers [13]. Dropout, as the name implies, deactivates neurons in layers with probability p (set to 0.1 in these tests) in order to prevent overfitting of the network.

4.3 Experiments

Some results are presented here regarding the potential applications of this type of models in two different experiments.

- Experiment 1 utilizes the background samples of the `t` dataset for training. The `b` dataset is used as a test set.

- Experiment 2 utilizes the background data of the `t` and `v` datasets for training. The `b` dataset is used as a test set.

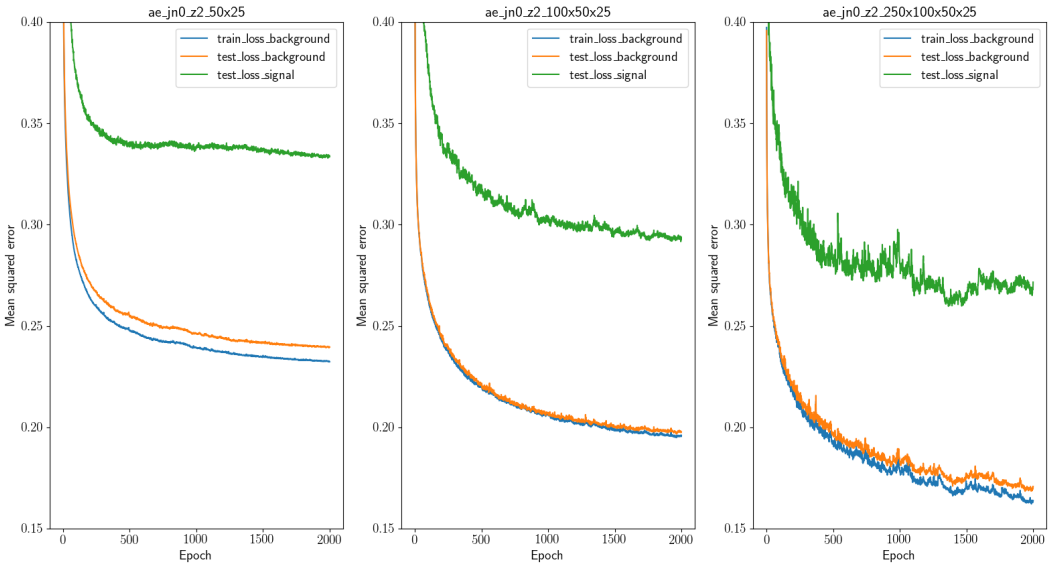


Figure 4.1: Loss function over epoch for 3 different autoencoder configurations. Despite loss being lower with larger layers, training is more unstable as can be seen in the jittering of the loss function. The latent representation was bi-dimensional, the learning rate was set to 1×10^{-4} and it was trained on data with `PRI_jet_num` $= 0$. The leftmost plot uses two hidden layers for the encoder of dimensions [50, 25] and symmetrically for the decoder. The center plot has three hidden layers of dimensions [100, 50, 25]. The rightmost plot has four hidden layers of dimensions [250, 100, 50, 25].

Each model was trained for 1000 epochs for a computation time of approximately one hour for Experiment 1 and almost 3 hours for Experiment 2.

4.3.1 Latent variables

It is possible to plot the latent variables (also called *embeddings*) of the models.

The latent variables of the models are plotted in figs. 4.2 to 4.5. As expected, the embeddings for the variational autoencoder resemble a gaussian distribution due to the additional constraint added to the latent space. The embeddings of the variational autoencoder for Experiment 2 result in a lower latent loss on the test set, as the latent distribution resembles more the prior gaussian $\mathcal{N}(0, \boldsymbol{I})$ compared to training with less data.

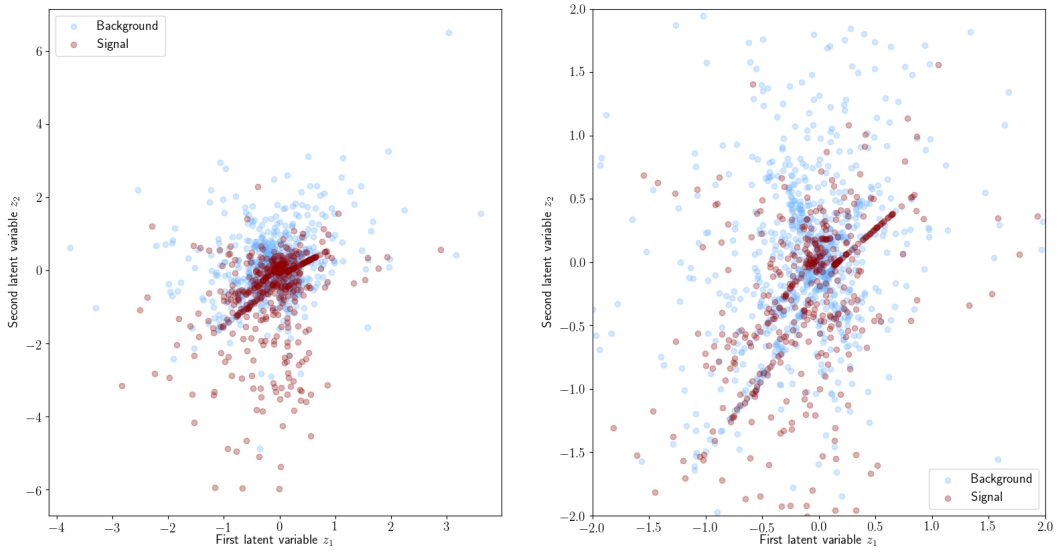


Figure 4.2: The embeddings for the autoencoder on the test set for Experiment 1. It is not possible to sample from this latent space. The plot on the right is a zoom in $[-2, 2]$.

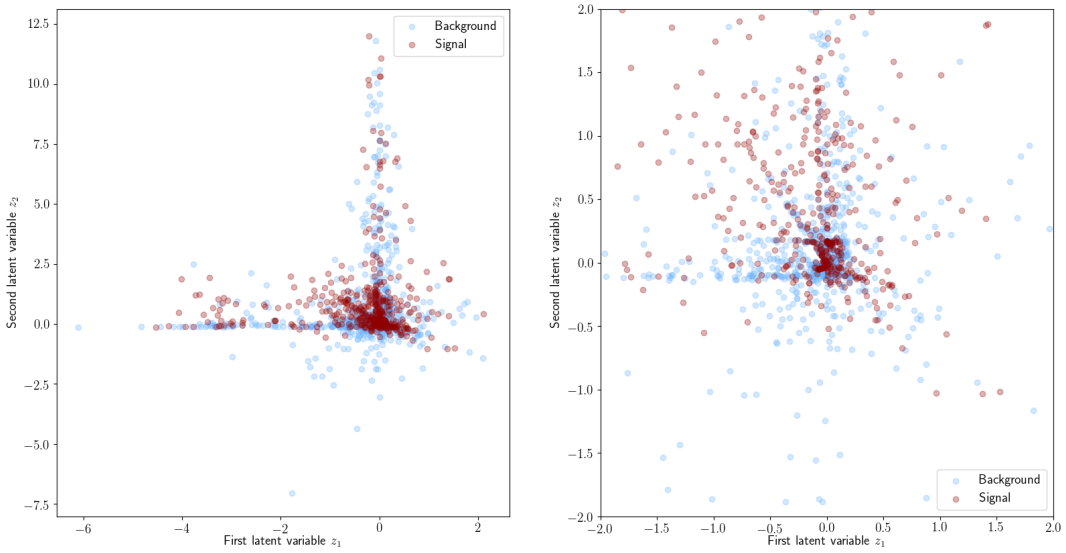


Figure 4.3: The embeddings for the autoencoder on the test set for Experiment 2. It is not possible to sample from this latent space. The plot on the right is a zoom in $[-2, 2]$.

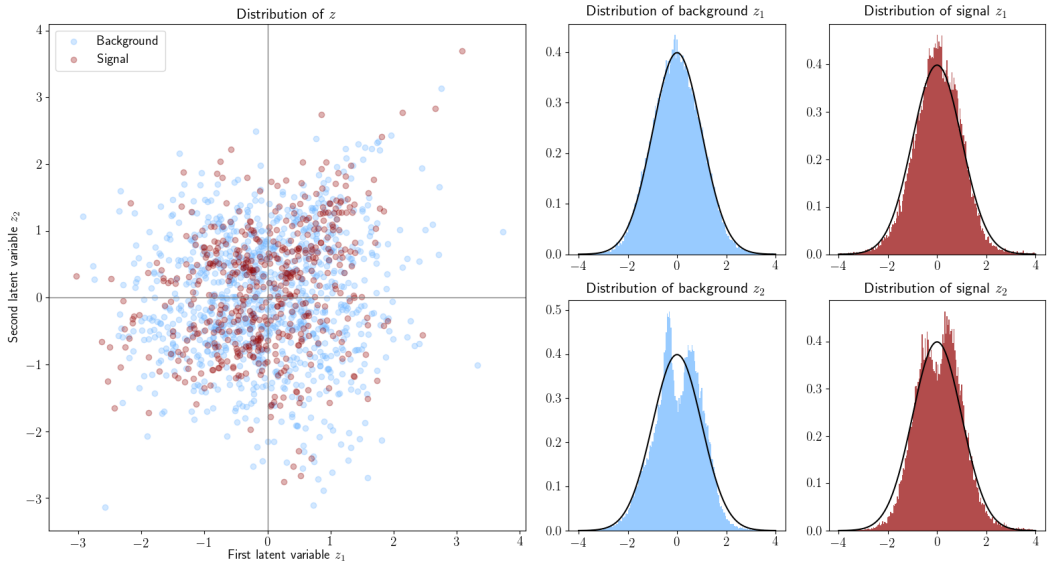


Figure 4.4: The embeddings for the variational autoencoder on the test set for Experiment 1. On the left are histograms for the distributions of the two dimensions (z_1 is the x axis and z_2 is the y axis).

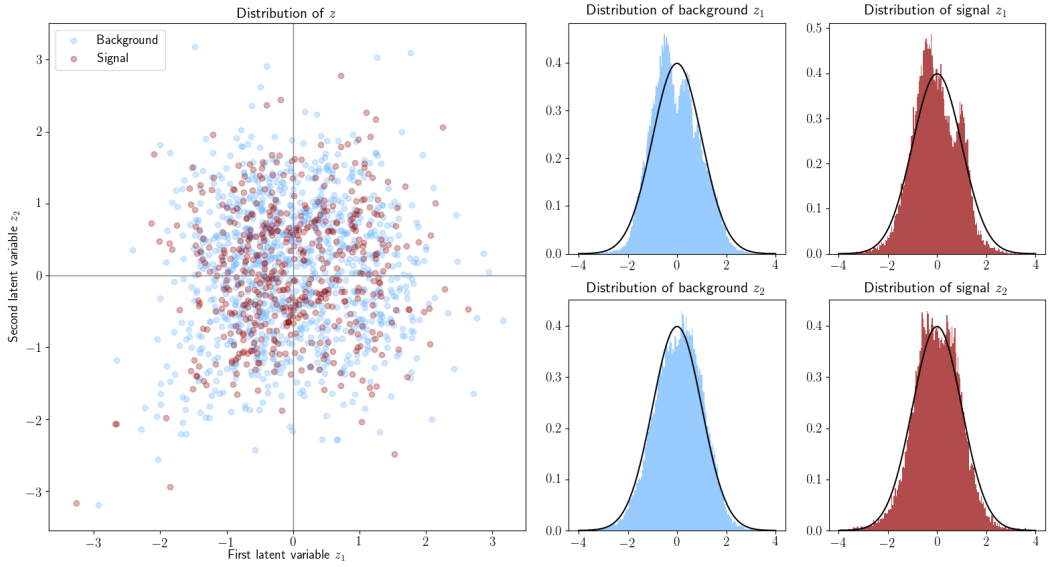


Figure 4.5: The embeddings for the variational autoencoder on the test set for Experiment 2. The gaussian distribution of the latent variables is more stable thanks to training with additional data.

4.3.2 Calculating the threshold for anomaly detection

Anomaly detection methods require a cutoff or threshold t in order to define what is and is not an anomaly. In the unsupervised setting of this project, we hypothesize a higher reconstruction error of signal data (compared to background), but as signal data is not used except in testing to calculate metrics, the possible thresholds have to be calculated on the training data. The histograms in figs. 4.6 and 4.7 show distributions of the reconstruction errors for training and test data. It is possible to see that as the number of jets increase, it is more difficult for the autoencoders to reconstruct the samples and at the same time, the gap between the reconstruction error for signal events and background

events is narrower.

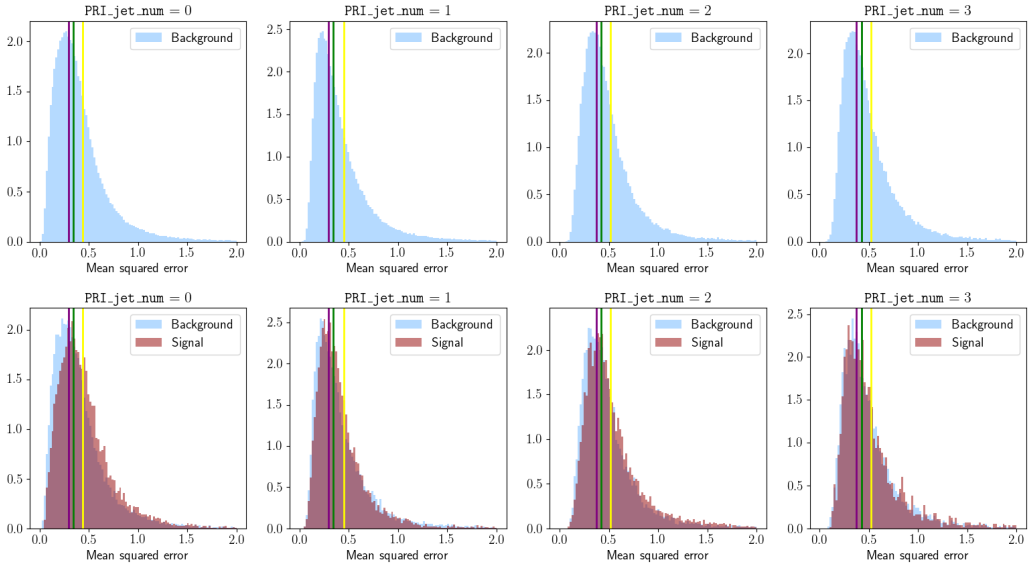


Figure 4.6: The first row show normed histograms of reconstruction error of background training data based on the number of jets for an autoencoder used in Experiment 2. The x axis is the value of reconstruction error. The purple, green and yellow vertical lines represent respectively the rank 40 percentile (approximately the mode), the median and the mean calculated on the training data. The second row shows the distribution of reconstruction error for the test data.

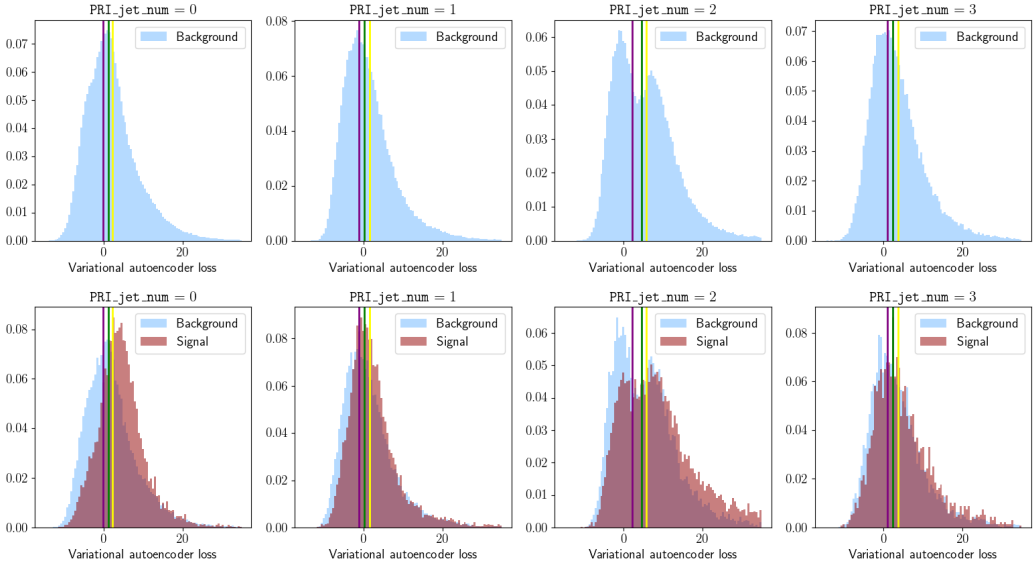


Figure 4.7: The same plots are shown here for the variational autoencoder. The x axis is the loss for the variational autoencoder. The error distribution for signal events is more different compared to the classical autoencoder.

4.3.3 Evaluation

The models are compared in tables 4.3 to 4.6. Each row is calculated respectively on the rank 40 percentile, the median and the mean described above. Note that the AMS metric, which is reported

as it was used in the original Kaggle challenge, is probably not suited to this kind of model. In fact, a simple model that classified everything as signal would achieve an AMS of 1.079. The results show how additional data improves the metrics and highlights how the variational autoencoder, in addition to being a generative model, achieves much better results compared to the classical autoencoder when fewer samples are available. It also has better metrics overall.

AMS	accuracy	precision	recall	f1 score	MCC
0.862	0.491	0.363	0.659	0.468	0.0610
0.778	0.521	0.365	0.555	0.440	0.0545
0.618	0.569	0.368	0.370	0.369	0.0421

Table 4.3: Metrics on the test set for experiment 1 for the autoencoder.

AMS	accuracy	precision	recall	f1 score	MCC
0.914	0.495	0.367	0.673	0.475	0.0738
0.840	0.524	0.370	0.568	0.448	0.0661
0.686	0.570	0.372	0.386	0.379	0.0505

Table 4.4: Metrics on the test set for experiment 2 for the autoencoder.

AMS	accuracy	precision	recall	f1 score	MCC
1.046	0.520	0.392	0.745	0.514	0.147
0.971	0.551	0.400	0.643	0.494	0.139
0.914	0.565	0.402	0.573	0.473	0.127

Table 4.5: Metrics on the test set for experiment 1 for the variational autoencoder.

AMS	accuracy	precision	recall	f1 score	MCC
1.047	0.520	0.392	0.748	0.515	0.149
0.990	0.552	0.402	0.647	0.496	0.143
0.932	0.575	0.410	0.567	0.476	0.139

Table 4.6: Metrics on the test set for experiment 2 for the variational autoencoder.

5 Conclusions

This work has shown possible uses of deep learning in unsupervised methods in difficult fields such as high energy physics. Future work could apply the models prototyped in this work for example in other tasks which deep learning excels at, such as computer vision which has been used before in HEP for classification of jet images [3].

Simple models have been tested on data with mediocre results, possibly due to the difficult dataset examined. However, it has been shown that more data leads to more stable latent representations. Having access to a simulator which could provide additional amounts of data could have improved the results of this work. Note that as autoencoding models learn the distribution over the input data, a very large amount of events (which is one of the strengths of deep learning) in the correct proportions might have helped in classifying signals as anomalies. The features used in the challenge and this work are for the major part "engineered" features: high energy physics used those features in order to improve classification potential. One of the major promises of deep learning is removing the need to manually create features, as the network will learn them through the various layers and transformations. For this reason, it would be interesting to test these techniques on the raw data from the detectors.

The work done in this thesis can be used to implement anomaly detection pipelines through generative models in other datasets related to HEP, or more generally in other fields. All the work done is available on my public repository (`https://github.com/Illedran/DL-HEP`) on GitHub. The code is written in Python using the TensorFlow and Keras libraries. Documentation to reproduce my results is described in the repository, along with detailed information about other libraries used and their versions in the `requirements.txt` file.

Bibliography

[1] Claire Adam-Bourdarios, Glen Cowan, Cecile Germain, Isabelle Guyon, Balazs Kegl, and David Rousseau. The higgs boson machine learning challenge. In *Journal of Machine Learning Research (JMLR): Workshop and Conference Proceedings*, volume 42, pages 19–55, 2015.

[2] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[3] Josh Cogan, Michael Kagan, Emanuel Strauss, and Ariel Schwartzman. Jet-images: computer vision inspired techniques for jet tagging. *arXiv preprint arXiv:1407.5675*, 2014.

[4] Luke de Oliveira, Michela Paganini, and Benjamin Nachman. Learning particle physics by example: Location-aware generative adversarial networks for physics synthesis. *arXiv preprint arXiv:1701.05927*, 2017.

[5] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

[6] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[7] Daniel Guest, Julian Collado, Pierre Baldi, Shih-Chieh Hsu, Gregor Urban, and Daniel Whiteson. Jet flavor classification in high-energy physics with deep neural networks. *Physical Review D*, 94(11):112002, 2016.

[8] Samuel C P Hall and Andrey Golutvin. *Searching for beyond the Standard Model physics using direct and indirect methods at LHCb*. PhD thesis, Imperial Coll., London, 2015. presented 10 Jul 2015.

[9] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[10] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[11] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[12] Evan Racah, Seyoon Ko, Peter Sadowski, Wahid Bhimji, Craig Tull, Sang-Yun Oh, Pierre Baldi, et al. Revealing fundamental physics from the daya bay neutrino experiment using deep neural networks. *arXiv preprint arXiv:1601.07621*, 2016.

[13] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.