POLITECNICO DI TORINO

Faculty of Engineering

Master of Science in Ingegneria Matematica

Master Thesis

# Variational Autoencoder for unsupervised anomaly detection


**Advisor:**
Prof. Paolo Garza

**Candidate:**
Francesco Lupo

**Company tutors**
**ML Reply**
Giorgia Fortuna

March 2019

To my loving family and my girlfriend, who supported during these years.

# Summary

The variational autoencoder is a generative model that is able to produce examples that are similar to the ones in the training set, yet that were not present in the original dataset. While this model has many use cases in this thesis the focus is on anomaly detection and how to use the variational autoencoder for that purpose. In the first part various state of the art anomaly detection algorithms are presented, in the second part the structure and the functioning of the variational autoencoder are presented, along with a comparison with the classic autoencoder. The details on how to exploit the variational autoncoder as an anomaly detection tool are also described in this part. In the third part experiments carried out with different datasets and different architectures are shown. In the last part a new use case is proposed, in particular on how to use the variational autoencoder to perform semantic novelty detection in a natural language processing context.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Anomaly detection has been a classic problem in machine learning with many applications. It can be used in order to spot faulty parts in a production chain, detect damaged packages, fraudulent transactions in a banking system and even cancerous cells in a biopsy.
While it may look like a specific application of the classic binary classification problem in most of these cases it's not useful to proceed in that fashion in order to solve the problem. Anomalous data is usually not available in big quantities, making the two classes highly unbalanced, which means a binary model will be hard to train. In other applications anomalous data could be completely absent in the training set, since one may want to detect anomalies that did not occur yet.

## 1.1 Problem Definition

Anomaly consists in spotting data that is supposedly produced by a generative process different by the one used for the normal data.
Given a dataset

$$x_1, x_2, ..., x_n \in \mathbb{R}^p$$

There are three possible scenarios:

- Supervised: the normal and anomalous data are labelled accordingly
- Clean: data only consists only of normal data
- Unsupervised: data consists of a mixture of normal and abnormal data

In the unsupervised scenario we will assume that our training data follow a contamination model

$$x_1, \ldots, x_p \sim (1-p)f_0 + pf_1$$

where $f_0$ is the distribution of the normal data and $f_1$ is the distribution of the anomalous data, and $p$ is reasonably small. We can note that if $p$ is very small we would fall back to the clean setting.

## 1.2 Previous work

This section describes the state of the art algorithms used in order to address the anomaly detection problem.

- Support vector machines for novelty detection [17]
- Kernel PCA for novelty detections [10]
- Isolation forest [14]
- Local outlier factor for anomaly detection [5]

### 1.2.1 One class SVM for anomaly detection

This model is used in an unsupervised setting, where the training data consists of both normal and non normal data.
Support vector machines are a very popular classification algorithm used in supervised learning. SVMs are a generalization of support vector classifiers.
Let's suppose our data has two labels $+1, -1$ and that is linearly separable. We wish to find the best hyperplane that divides our data in two regions. The first problem we need to solve is defining "the best" hyperplane, since in many cases there are infinite valid ones. One way to proceed is by defying the best hyperplane as the one that has the biggest margins, meaning that maximizes the perpendicular distance between the hyperplane and the points in the dataset. The vectors that are closest and equidistant from the boundary are called support vectors [11].
This method has the advantage that the decision boundary does not depend on the points that are far from the hyperplane, but adding a new point close to the

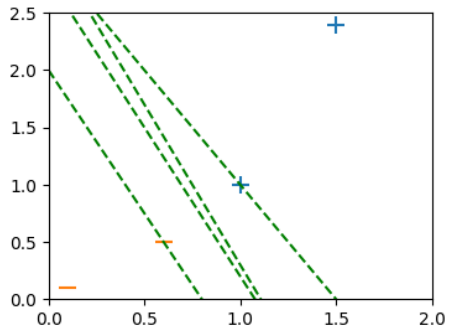



Figure 1.1: Some examples of the possible decision boundaries

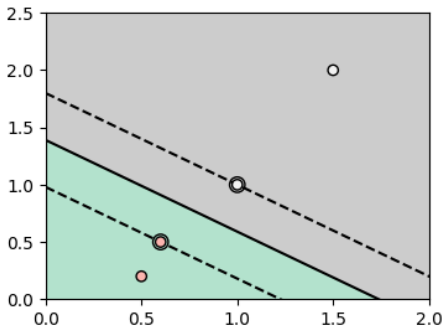



Figure 1.2: decision boundary found by linear svm

decision boundary changes the latter significantly, meaning that it's not robust and prone to overfitting.

Once we found a possible hyperplane we would be able to classify a new example $\hat{x}$ in our feature space $R^p$ simply by checking in which side of the plane it lays.

$$\beta_0 + \beta_1\hat{x}_1 + \cdots + \beta_p\hat{x}_p > 0 \quad \text{then} \quad \hat{y} = +1$$

$$\beta_0 + \beta_1\hat{x}_1 + \cdots + \beta_p\hat{x}_p < 0 \quad \text{then} \quad \hat{y} = -1$$

Which translates to

$$\hat{y}(\beta_0 + \beta_1\hat{x}_1 + \cdots + \beta_p\hat{x}_p) > 0$$

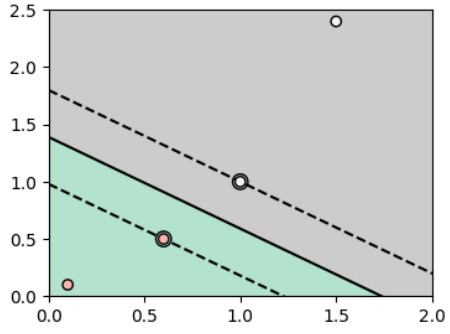



Figure 1.3: The decision boundary doesn't change if we move a point that was outside the margin such that it stays outside the margin.

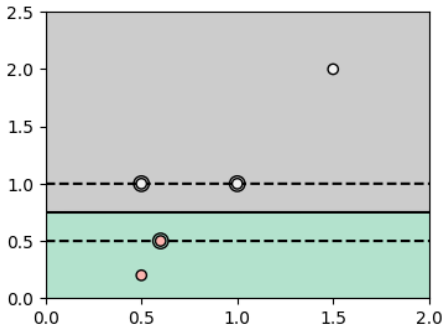



Figure 1.4: The decision boundary changes significantly if we add a point close to the boundary

In order to find the margin and the weights of the hyperplane we need to solve the following optimization problem [11]

$$\max_{\beta_0,\dots,\beta_p} M$$

$$\text{subject to} \quad \sum_{j=0}^{p} \beta_j^2,$$

$$y_i(\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p) > M \quad \forall i = 1, \dots, n$$

Another limitation we need to overcome is that we still need our data to be linearly separable, which is a very strict constraint.
We can partially solve these issue by adding the concept of a soft margin. That means that we let the points in our dataset cross the decision boundary, but in order to classify most of the observations correctly we need to introduce a cost with every misclassified example.

$$\max_{\substack{\beta_0,\dots,\beta_p \\ \epsilon_1,\dots,\epsilon_n}} M$$

$$\text{subject to} \quad \sum_{j=0}^{p} \beta_j^2,$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip}) > M(1 - \epsilon_i) \quad \forall i = 1, \dots, n$$

$$\epsilon_i \geq 0, \sum_{i=0}^{n} \epsilon_i < C$$

Even with soft margins not many classification problems can be solved by linear decision regions, that's why we need non-linear kernels.
A key finding of SVMs is that the decision boundaries weights only depends by the dot product between $x$ and the support vectors $x_i$ [11].

$$f(x) = \beta_0 + \sum_{x_i \in SVs} \alpha_i \langle x, x_i \rangle$$

If we replace the dot product with a more general function the we will call *kernel*, and we will be able to find more complex decision boundaries.
Common kernel functions that are widely used are:

$$K(x_i, x_{i'}) = x_i^{\intercal} x_{i'} \text{ Linear Kernel}$$

$$K(x_i, x_{i'}) = exp(-\gamma \sum_{j=1}^{p}(x_{ij} - x_{i'j})^2)) \text{ Radial basis function}$$

$$K(x_i, x_{i'}) = (x_i^\intercal x_{i'} + c)^d \text{ Polynomial kernel of d-degree}$$

Let's now see how this supervised technique can be used in an unsupervised way for anomaly detection.
Recalling the scenario in which we are working we are assuming that our data contains both normal and anomalous data

$$x_1, \ldots, x_p \sim (1-p)f_0 + pf_1$$

where $f_0$ is the distribution of normal data and $f_1$ is the distribution of the anomalous data.
The intuition behind the one class SVM is that the normal data will fall in the region predicted by the algorithm, while the anomalous data will fall outside of the margins. We need to add a few more constraints to our data for the one class SVMs to work properly. The first one is that $p$ should be small, while the second one is that $f_1$ should have a much larger support of $f_0$, otherwise the anomalous data will cluster and the SVM will incorporate that data in the one-class. Lastly, the support of the distributions should have little to no overlap.

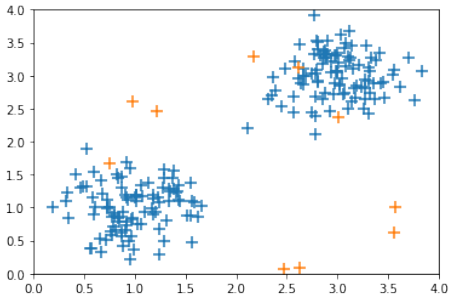



Figure 1.5: Synthetic dataset

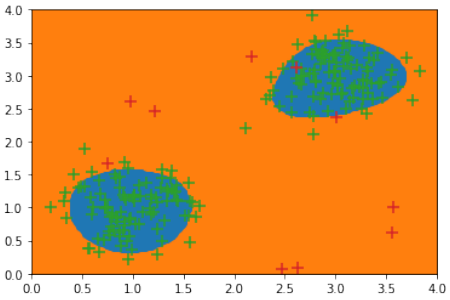



Figure 1.6: SVM decision boundary

### 1.2.2 Kernel PCA for novelty detection

This method is used in a "clean" setting, where only normal data is used at training time.

#### Principal component analysis

Principal component analysis is a dimensionality reduction technique. It defines a new set of dimensions where each of the new dimension is a linear combination of the original features. The first new dimension would be

$$Z_1 = \phi_{11}X_1 + \phi_{21}X_2 + \cdots + \phi_{p1}X_p \qquad \text{con} \sum_{j=1}^{p} \phi_{j1}^2 = 1$$

While the number of new dimensions can be as high as the original feature space we need to find only the most relevant ones. In order to do this we are going to identify the direction in which the data has maximum variance. While the first component

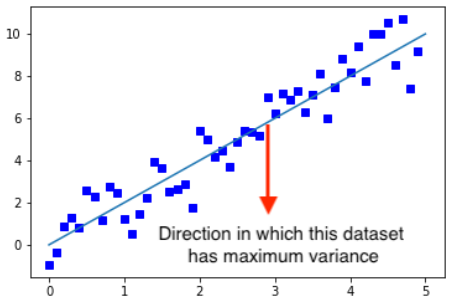



Figure 1.7: Original Dataset

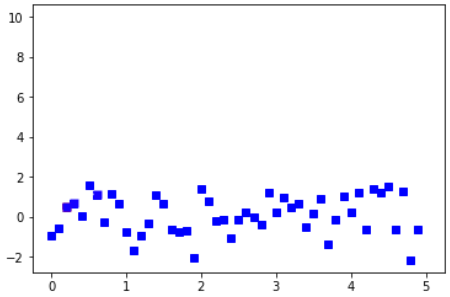


Figure 1.8: Dataset after PCA transformation

found by PCA is the one that explains the most variance, the second component has to be perpendicular to the first one, and should be the one that explains the most variance not considering the first one.
While all of these components could be found by solving an optimization problem it turns out that exploiting the covariance matrix is the better way to proceed.

$$C = \frac{1}{N}X^\intercal X$$

The eigenvectors of the covariance matrix will form the basis of the the new vector space, while the respective eigenvalues represents the amount of variance explained

through that axis. This means that the first component is the eigenvector that has the biggest eigenvalue.

$$V^{-1}CV = D$$

Where $V$ is the matrix of eigenvectors, and D is a diagonal matrix where the eigenvalues are.

#### Kernel PCA

The main disadvantage of PCA is that the new components are a linear combination of the old ones, which means it's not going to perform well for high dimension non linear dataset. For this reason kernel PCA has been introduced [20].
Instead of using our dataset we are going to map it to a higher dimensional space using a non linear function called kernel.

$$x_i \in \mathbb{R}^p \rightarrow \Phi(x_i) \in \mathbb{R}^M$$

$$M >> p$$

After this mapping simple PCA could be performed, but since it would be very computationally expensive a new kernel matrix is used to carry out the computation, achieving the same result (details in [20]).
Now that we have a new feature space we need to use it to identify anomalies [10]. At training time we are going to feed our model with only non anomalous data, and we will then decide how many components to keep to compute the contracted version of the dataset. At test time we will compute the components of our test examples and then we are going to reconstruct them. The anomaly score will be the reconstruction error in our new feature space.
A common kernel is

$$k(x, y) = exp\frac{-||x - y||^2}{2\sigma^2}$$

### 1.2.3 Isolation forest

This model is used in a scenario similar to the one class SVM's one, specifically in an unsupervised setting. The isolation forest takes a different approach from the one class SVM, since instead of grouping normal data it tries to isolate the anomalous data. The isolation forest basic component is the isolation tree, which is a simple

binary tree where at each node $T_i$ both the feature and threshold for our splitting rule are picked randomly. An existing node stops generating children if and only if there is only one example following the splitting rule for that specific path (meaning the example has been isolated) or a maximum height has been reached. This means that at the end of the training process we will have a completely overfitted random classification tree, that can be used for anomaly detection purposes. The main intuition of this algorithm is that if an example is anomalous it will be isolated after few cuts in the feature space, which translates having a low height in the isolation tree. This kind of score is non straight-forward but it has been represented by the following formula [14]

$$s(x, n) = 2^{-\frac{E(h(x))}{c(n)}}$$

$$\text{where} \quad c(n) = 2H(n-1) - 2(n-1)/n$$

Where n is the cardinality of our training data, $H(n)$ is the harmonic number and $h(x)$ is the height of the example $x$ in the isolation tree. We notice that we use the expected value of the height, since we actually will be using a forest of trees where each tree has been trained with a subset of the training data.
The score $s(x, n)$ will be close to one for anomalies, while it will be closer to zeros for normal data.

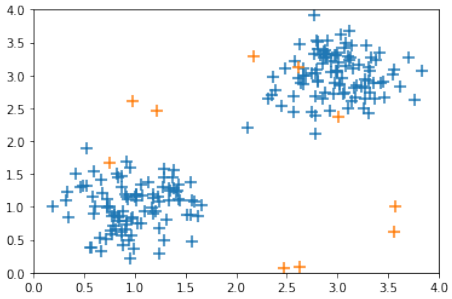



Figure 1.9: Synthetic dataset

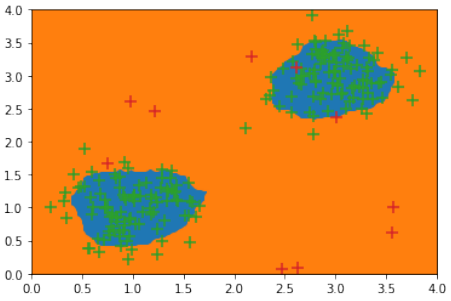



Figure 1.10: Isolation forest decision boundaries

### 1.2.4 Local outlier factor for anomaly detection

The local outlier factor (LOF) [5] is a density based anomaly detection algorithm. The factor itself is a measure of how outlying a specific point is: $lof \approx 1$ not an

outlier, $lof \gg 1$ for outliers.
It's a local method since only a neighborhood of each point is considered in order to compute its LOF, and should be used in an unsupervised setting.
We are going to need to introduce some preliminaries definitions in order to understand how it works. The first one is the k-distance of a point, which is the distance of the k-th clostest point. The second one is the reachability distance.

$$reach - distance(p, q) = max(k - distance(q), d(p, q)$$

where $d(p, q)$ is the distance between two points. If $p$ is in the radius defined by the k-distance around $q$ the reach distance is simply the k-distance, otherwise it will be the standard distance between two points.
We now are going to define the local reachability density of a point. The $lrd$ is the inverse of the average of all the reach-distances between the point and all of its k-neighbours.

$$lrd(p) = \frac{k}{\sum_{q \in neighbours} reach - distance(p, q)}$$

Now we are going to define the local outlier factor itself. The $lof(p)$ is the average between the ratios between every point in the neighborhood and the point $p$.

$$lof(p) = \frac{\sum_{q \in neighbours} \frac{ldr(q)}{ldr(p)}}{k}$$

Now that we defined the $lof$ we only need to decide a threshold for deciding when a point can be defined as an outlier, but it has to be specific for every dataset.

Isolation forest has an overall good performance for clustered data, and is not susceptible to the addition of useless features. Since LOF is density based it will have problems with dataset with a large amount of features, since in those spaces the distance between two points is less meaningful (this problem is also known as curse of dimensionality) which means that it will not perform good with images. KPCA and SVM are both non-linear models and can work with simple images.

## 1.3 Neural Networks and Generative Models

All of the models presented in the previous section will have troubles if applied to dataset with a large amount of features, like images. These complex datasets are

historically being treated with deep and convolutional neural networks. For this reason we are now introducing neural networks and generative models (a specific application of NNs) and in the second chapter we will exploit there models in order to build an anomaly detector.

### 1.3.1 Perceptron

Neural networks are a popular machine learning technique, they can be used for both regression and classification since they can be considered function approximators. The building block of neural networks are perceptrons, a computational unit that mimics how the biological neuron works.

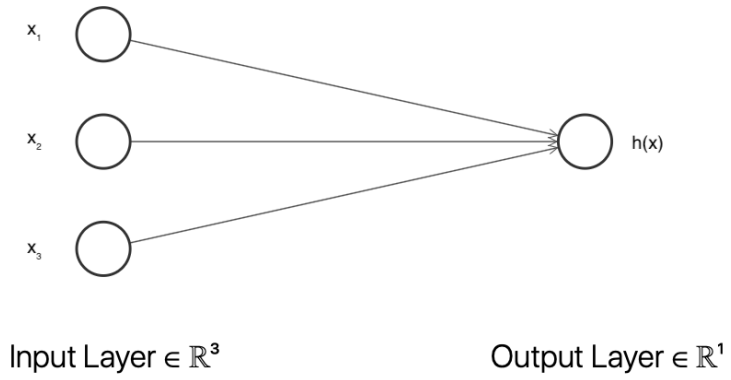



Figure 1.11: Perceptron architecture

where $h_\theta$ is a non linear function called activation function.

$$h_\theta(x) = h(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3)$$

which can be written in vector notation adding the element $x_0 = 1$ to $x$, so that both $x, \theta \in \mathbb{R}^{p+1}$.

$$h_\theta(x) = g(\theta^\intercal x) = g(z)$$

In this formulation $\theta_0$ is called bias, while $\theta_1, \ldots, \theta_p$ are called weights. The most used activation functions are the Sigmoid function (or logistic function) and the ReLu (Rectified Linear Unit) function.

$$g(z) = \frac{1}{1 + e^{-z}} \text{ Sigmoid function}$$

$$g(z) = \begin{cases} 0 \text{ for } z < 0 \\ z \text{ for } z \geq 0 \end{cases} \quad \text{ReLu function}$$

If the perceptron uses the Sigmoid function as activation it is actually able to perform logistic regression, since the Sigmoid function represents the probability of one example of belonging to one class. More specifically, given a dataset

$$D = \{x^{(i)}, y^{(i)}\} \text{ with } x^{(i)} \in \mathbb{R}^p, y^{(i)} \in \{0,1\}, i = 1, \ldots, n$$

we want the output of our perceptron to be the probability of belonging to class 1. The only things we can change in our model are the weights $\theta$, and one option would be to do it minimizing the Mean Squared Error (MSE) between the prediction and the actual class

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} (y^{(i)} - h_\theta(x^{(i)}))^2$$

but since the MSE is not a convex function it is hard to optimize. In order to solve this problem a modified convex function has been introduced

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} cost(y^{(i)}, h_\theta(x^{(i)}))$$

$$cost(y, \hat{y}) = \begin{cases} -\log \hat{y} \text{ if } y = 1 \\ -\log(1 - \hat{y}) \text{ if } y = 0 \end{cases}$$

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

Looking at Figure 1.12 we can interpret this new cost function: if the predicted probability of one example $\hat{y}$ is close 0, but the actual class is 1 then that example will contribute to the total cost with a high value, while if the predicted probability was rightfully close to 1 then that example would not contribute to the cost at all. Since the cost function for examples with $y = 0$ is symmetric to the one for examples with $y = 1$ we can do the same kind of reasoning for the other part of the cost function.
Now that we have a model and a cost function to minimize we only need to find a way to change the values of the weights such that they will converge to a solution that gives us the expected output. The most common of these optimization algorithms is Gradient Descent.

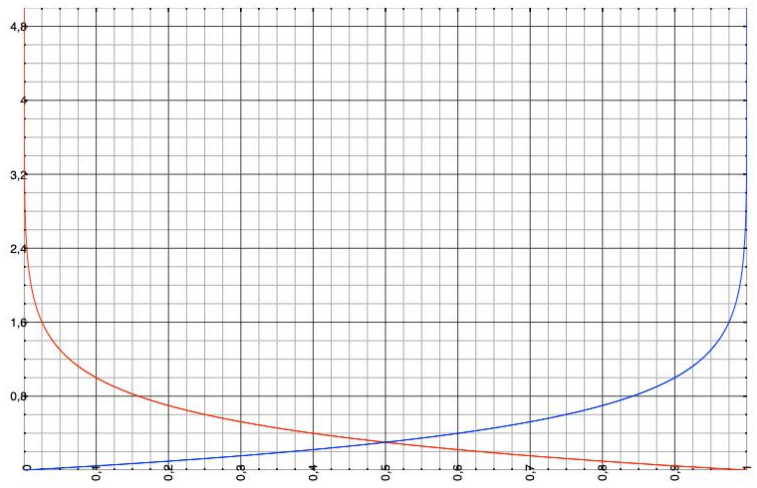



Figure 1.12: in red: cost for $y = 1$
in blue: cost for $y = 0$

### 1.3.2 Gradient Descent

After finding the appropriate cost function the problems becomes

$$\min_{\theta} J(\theta)$$

which can be solved exploiting the derivative of the cost function.
Suppose our cost function to be strictly convex and that we can compute the derivative of this function with a closed form. If we start from a random point in the function domain then we could understand in which direction we should move in order to get closer to the minimum by looking at the derivative.

$$\textit{repeat until convergence}\{\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)\}$$

Where *alpha* is the learning rate, which represents how big of a step we are making at each iteration. A big learning rate implies faster training times but if too big can result in the algorithm not converging at all.

### 1.3.3 Neural Networks

In order to form a neural network we add hidden layers between the input layer and the output layer. Each one of the input nodes is connected to each node of the first hidden layer, but the weights are going to be different for each node of the hidden layer in order to compute different intermediate features that will help our classification.

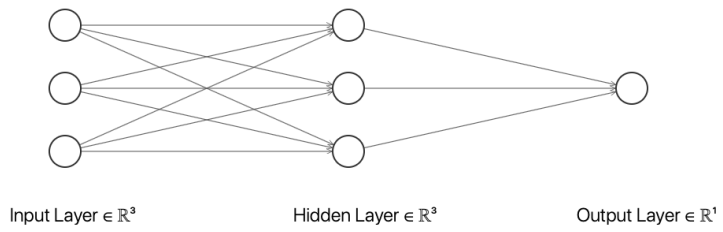



Figure 1.13: Neural Network architecture

Let us define the notation of our neural network.

- $a_i^{(j)}$ is the activation unit $i$ of the layer $j$
- $\theta^{(j)}$ will now be a matrix of weights controlling the function mapping between layer $j$ and layer $j+1$

We can compute the value of the activation function of the middle layer as such

$$z_1^{(2)} = \theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \cdots + \theta_{1n}^{(1)} x_n$$
$$a_1^{(2)} = g(z_1^{(2)})$$
$$z_2^{(2)} = \theta_{20}^{(1)} x_0 + \theta_{21}^{(1)} x_1 + \cdots + \theta_{2n}^{(1)} x_n$$
$$a_1^{(2)} = g(z_1^{(2)})$$

Which in matrix notation becomes

$$z^{(j)} = \theta^{(j-1)} a^{(j-1)}$$
$$a^{(j)} = g(z^{(j)})$$

Where $g$ is an activation function.

### 1.3.4 Backpropagation

Since we now have a more complex architecture with different layers we are going to need a more complex algorithm than gradient descent in order to understand how to change the weights of each layer.
Let's consider a problem where the cost function $J(\theta)$ is given by the mean squared error

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - h_\theta(x^{(i)}))^2$$

Each of the example contributes to the final value of the cost of the entire network, but let us focus on the contribution of only one example in the training data, let's call it $J_0$

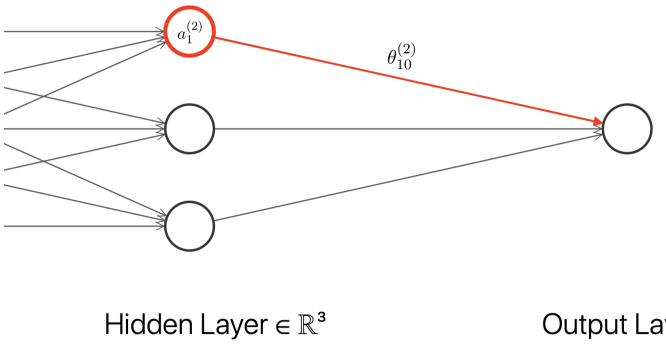
$$J_0 = (y^{(i)} - a_1^{(3)})^2$$

Let's suppose that the current value of $J_0$ is not the expected one, this means that we should change the weights of the network so that it will give us the expected output, in particular $a_1^{(3)}$

$$a_1^{(3)} = g(z^{(3)})$$

$$z_1^{(3)} = \theta^{(2)}a^{(2)}$$

The first weights we can change are the ones that are directly connected to the output, let's look at one in particular and compute the derivative of $J_0$ with respect to it, exploiting the chain rule



$$\frac{\partial J_0}{\partial \theta_{10}^{(2)}} = \frac{\partial z^{(3)}}{\partial w^{(3)}} \frac{\partial a^{(3)}}{\partial z^{(3)}} \frac{\partial J_0}{\partial a^{(3)}} = a_1^{(2)} g'(z^3) 2(a^{(3)} - y)$$

We can now use this derivative to update the value of $\theta_{10}$. The piece of cost function $J_0$ is not only sensitive to the weights but also to the activation value of the layer before, but we can't directly change that. What we can do is change the values of the weights that produced that activation value. In order to change those values we need to compute the derivatives of $\theta^{(1)}$ in a similar fashion as before.
This process has to be done for every example in our training dataset and for every weight in the network for how many iterations it takes the algorithm to converge. For very big networks with big dataset this process can be very slow, but it can be accelerated using batches: at every iteration only a representative subset of the dataset is taken and used to update all the weights in the network.

### 1.3.5 Convolutional Neural Network

When dealing with images neural networks with only fully connected layers can be very hard to train because the amount of parameters involved. For example for a black and white image with size $28 \times 28$ would be represented with an array of 784 elements. If we wanted to add a layer with with 200 nodes after the input layer the matrix would have size $784 \times 201$, for a total of 157584 parameters to train.
This is one of the reasons the Convolution operation has been introduced in neural networks.

#### The Convolution operation

The convolution operation can be defined between to real value functions but we are going to need it only for discrete functions. For two functions $f$ and $g$ defined over a set of integers $i \in I$ we have

$$(f * g)[i] = \sum_m f[m] g[i - m]$$

which for two dimension discrete functions with indexes $i \in I, j \in J$ becomes

$$(f * g)[i, j] = \sum_m \sum_n f[m, n] g[i - m, j - n]$$

The function $g$ is called kernel or filter. While this is the formulation used in signal processing in machine learning the actual operation that is done is cross-correlation, even if everybody refers to it as convolution. The only difference is that in cross-correlation is the minus sign in the kernel.

$$(f \star g)[i,j] = \sum_m \sum_n f[m,n]g[i+m, j+n]$$

We are going to refer to cross-correlation as convolution from now on.
When applied to images the convolution is performed with a filter whose dimension are smaller than the unfiltered images. At every step of the summation the filter is placed on top of a specific region of the image (starting from the upper left corner) and element wise matrix multiplication is performed, then all the entries of the resulting matrix are summed. At the next step the filter is moved of one pixel to the right and so on.
At training time the values of the filter is what is to be determined, calculating the gradients and performing backpropagation. From a $n \times n$ image and a $f \times f$ filter a $n - f + 1 \times n - f + 1$ image will be produced.
A useful parameter that can be added to the normal convolution to make it perform faster is the stride. With a stride of 1 normal convolution is performed, with a stride of 2 or more the filter is moved of 2 or more pixel at every step of the summation.

#### Transposed convolution

Transposed convolution (often referred as deconvolution) is used to obtain an image with larger dimensions than the input one.
In order to accomplish that a fractional stride is used, which means every row and every column of the input image a row or column made of zeros is added. Convolution is performed as usual after this modification of the input image.

### 1.3.6 Generative models

Generative models are an unsupervised learning technique that aims at generating new and unseen examples that are similar to the ones in the dataset. For a dataset with only one numeric feature a simple generative model could consist in simply fitting a known distribution to our data and then sampling from it. This method can

work for higher dimension as far as we have a good model for learning an explicit probability density for our data, which is a very difficult task for high dimension data (like images).
There are more complex models were the probability density is explicit, Neural Autoregressive Distribution Estimation (NADE), Masked Autoencoder for Distribution Estimation (MADE), PixelRNN and PixelCNN (the last two are specifically designed for images).

### 1.3.7 PixelRNN

The main purpose of this architecture is to model the probability p(x) where x is a vector that describes an image, exploiting the product rule

$$p(x_1, x_2, \ldots, x_n) = \prod_{n=1}^{N} p(x_n | x_1, \ldots, x_{n-1})$$

The pixelRNN [19] model starts by generating the first pixel in the upper left-hand corner and then proceeds by generating the other pixels sequentially, where the every pixel influences directly the value of the pixel to its right and the pixel below him. This dependency is modeled using a recurrent neural network, more specifically using an LSTM layer (whose weights are shared for the generation of every pixel). This does not mean that the value of one specific pixel depends only on its direct neighbour since an RNN is able to retain information of long sequences. At training time the networks aims at assigning a probability $p(x)$ to every pixel maximizing the negative log likelihood. The major drawback of this approach is that sequential generation is very slow.

### 1.3.8 PixelCNN

The pixelCNN [19] works similarly to the pixelRNN, in fact it also starts by the upper left-hand corner to generate the image, but it models the dependencies between the current pixel and the previous ones using a convolutional neural network. The pixel area used to model the current pixel is called "context region". In this case the dependency is models using a convolutional neural network with filter over the context region. This has the advantage of leading to faster training time (since

convolution can be parallelized) but generation needs still to be sequential, hence slow.

### 1.3.9 GAN

While the last two methods computed an explicit and tractable probability density the generative adversarial network (GAN) uses a completely different approach. There are two neural networks: the first one (called generator) takes as input random noise and outputs an image (that will hopefully be "realistic") while the second network (called discriminator) takes as input an image and classifies it as "fake" or "real" (outputting likelihood between 0-1 of real image).
The players are effectively playing a two player game, which means that for the model to work it is necessary to find a Nash equilibrium. The objective function of this minimax game is

$$\min_{\theta_g} \max_{\theta_d} [\mathbb{E}_{x \sim p_{data}} \log D(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D(G(x)))]$$

where $D(x)$ is the output of the discriminator when a real image has been given as input, while $G(x)$ is the output of the discriminator when a fake (meaning made by the generator) has been given as input. We can note that the discriminator wants to maximize the objective such that $D(x)$ is close to 1 when the input is real, while it wants to minimize $D(G(x))$ when the input is fake. The generator wants to do the exact opposite minimizing the objective, such that $D(G(x))$ is close to 1 (meaning that the discriminator has been fooled).
Different architectures have been used for both the generator and the discriminator, two examples are the simple fully connected layers, and the convolutional architecture. After training we can use the generator to network to produce new samples of our dataset, by giving random noise as input. It is interesting to note that if we take two generated images and their respective random noise we can create a path in the latent space that connects the two random noise, and generate images sampling from points laying in that path. In this way we are able to produce samples that gradually transition from the first generated image to the second one.
Another interesting property of the latent space $z$ where we sample from the random noise is that simple vector math works out. For example: if we take the vector $z$ that produces the image of a man with sunglasses, subtract to that a vector $z$ that

produces the image of a man, and sum to the results a vector $z$ that produces the image of a woman, then feed the resulting vector to the generator we will obtain the image of a woman with sunglasses.

# Chapter 2

# Variational Autoencoder

The variational autoencoder is a generative model with a probabilistic background. We are not going to use its generative capabilities but we are going to exploit its ability to model very complex distributions in a latent space in order to detect anomalies.

## 2.1 Preliminaries: Information Theory and KL Divergence

### 2.1.1 Information

In probability theory the information of an event represents how surprised you would be if that event would occur [3]. For example if you were in the middle of a very hot summer, the sentence "It's going to snow tomorrow" would carry a lot of information, while the sentence "It's going to be sunny tomorrow" would carry almost no information, since that outcome is very likely. More formally:

$$P(\text{"It's going to snow tomorrow"}) = p_{snow}$$

$$P(\text{"It's going to be sunny tomorrow"}) = p_{sunny}$$

Then the Information associated to those two events would be:

$$I(\text{"It's going to snow tomorrow"}) = \log_2 \frac{1}{p_{snow}}$$

$$I(\text{"It's going to be sunny tomorrow"}) = \log_2 \frac{1}{p_{sunny}}$$

If we assume that $p_{sunny} >> p_{snow}$ then the sentence "It's going to snow tomorrow" will carry more information than the sentence "It's going to be sunny tomorrow" which can be written as $I(\text{"It's going to snow tomorrow"}) > I(\text{"It's going to be sunny tomorrow"})$ (and can be easily proved using the properties of logarithms). In general, the Information of an event $E$ is written as:

$$I(E) = -\log_2 P(E)$$

The unit measure of information is called *bit*, when an event with probability 0.5 occurs we gain 1 bit of information since $\log_2 0.5 = 1$. It's also worth noting that the information of a certain event (an event with probability equal to one) is zero, consistently with our initial observations.

### 2.1.2 Entropy

Since the information describes only single events we need a new definition for describing probability distributions. Given a probability space $(\mathcal{X}, \mathcal{F}, P)$ we can define the entropy of a discrete random variable $X$ the expected value of the information.

$$H(X) = \mathbb{E}[I(X)] = -\sum_{x \in \mathcal{X}} P(x) \log P(x)$$

and can be interpreted as the average amount of information that a random variable carries. Let's compare the entropy of the outcome of a fair coin (we will call this r.v. $X_1$) and the entropy of the outcome of a coin where $P(head) = 0.8$ (we will call this r.v. $X_2$ ):

$$H(X_1) = -0.5 * \log_2 0.5 - 0.5 * \log_2 0.5 = 1$$

$$H(X_2) = -0.8 * \log_2 0.8 - 0.2 * \log_2 0.2 = 0.72$$

$X_1$ has the state of maximum uncertainty and has the greatest entropy, while $X_2$ has less uncertainty and also has a smaller entropy. The behavior of the entropy with respect to the probability of tossing head is shown in figure 2.1.

We can then conclude that Entropy is a measure of uncertainty. At the same time the entropy can be interpreted as how much information is in our distribution.

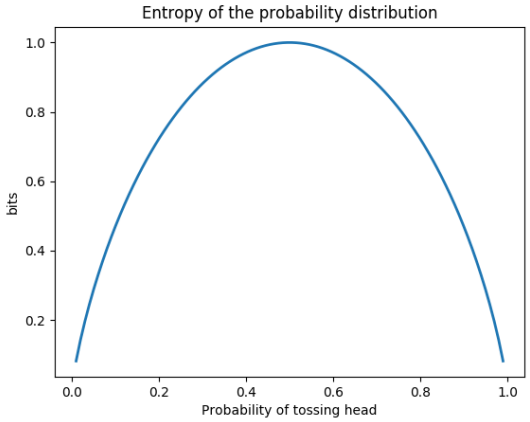



Figure 2.1: Plot of the entropy for a binomial distribution

### 2.1.3 Kullback-Leibler Divergence

In the next section we will need to approximate a complex probability distribution with a simpler (parametric) distribution, but in order to do this we need a measure of how much information is lost in the process. That is why we need to introduce the KL divergence. Given two discrete random variables $X$ and $Z$ defined over the same set $\{x_1, \ldots, x_n\}$ and their respective probability densities $p$ and $q$, the Kullback-Leibler Divergence can be expressed as:

$$\mathcal{D}_{KL}[p||q] = \mathbb{E}_{x\sim X}[\log p(x) - \log q(x)] = \sum_{i=1}^{n} p(x_i)[\log p(x_i) - \log q(x_i)]$$

I should be noted that this is not a symmetrical quantity since $\mathcal{D}_{kl}(p||q) \neq \mathcal{D}_{kl}(q||p)$ and that the divergence satisfies $\mathcal{D}_{kl}(p||q) \geq 0$ and $\mathcal{D}_{kl}(p||q) = 0$ if and only if $p(x) = q(x)$ [4] page 55

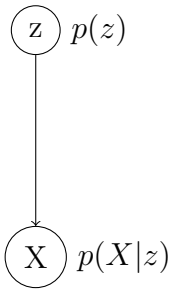
## 2.2 Probability Model

A Variational Autoencoder is a probabilistic generative model where the probability density $P(X)$ is modeled through a latent variable $z$. Our objective is to model $P(X)$ such that sampling from the distribution would lead to a convincing example of our dataset, yet that is not in the original dataset. The relation between the latent

variable and the original vector space is given by

$$P(X) = \int_z P(X|z)P(z)dz \tag{2.1}$$

Since the variational autoencoder can also be seen as a directed probabilistic graphical model, we can also express it as a graph:




Where $P(X|z)$ represents the probability distribution of $X$ conditioned on $z$. The main problem of 2.1 is that the integral over $z$ is intractable since the dimension of $z$ could be relatively large. We can obtain the posterior distribution of $z$ using Bayes theorem.

$$P(z|X) = \frac{P(X|z)P(z)}{\int_z P(X|z)P(z)dz} \tag{2.2}$$

The variable $z$ should represent latent information of the original dataset, if we take as an example the MNIST dataset of hand written digits [13] it would be desirable if each component of $z$ represented which digit is going to be generated by the model, the thickness of the stroke, or the skewness of the digit. On the other hand choosing and hard coding this information into the latent variable would not be feasible.

The VAE (Variational Autoencoder) takes another approach and assumes that $z$ has a know prior distribution $P(z)$ where each component has no easy interpretation. One choice might be having $z \sim \mathcal{N}(0, I)$.

One consequence of this model is that following a trajectory in the vector space of $z$ and sampling an example of $X$ for each point of the trajectory would lead to a series of images with similar characteristics, transitioning from a digit to another, as shown in the example below.

Even if we now have an easy way of sampling from the latent variable $z$ we still don't know how to sample from the original vector space of $X$, $\mathcal{X}$. In order to do that we need do build a function $f(z;\theta)$ where $f : \mathcal{Z} \to \mathcal{X}$. The easiest way to

Figure 2.2: Images generated following the path (1,1) $\rightarrow (1,-1)$

Figure 2.3: Images generated following the path $(1,-1) \rightarrow (-1,-1)$

model such function is to build a neural network and optimize over $\theta$ to suit our needs. Even if we now have both $P(z)$ and $P(X|z)$ we are still not able to compute the likelihood of our data $X$ since the equation 2.1 has an intractable integral. It is useful to note that the probability P(X) is very close to zero in most of the domain of X. Following the MNIST example, most of the domain does not look like a written digit, some of might look like a non-existing character, but most of it will look like random noise, and for all of those example P(X) has to be very close to 0. For this reason it is a good idea to sample from value of z that have a high probability of having produced an example X. In order to do this we introduce a new function, $Q(z|X)$, which given $X$ gives us a distribution over $z$ which represents the values that are likely to produce $X$. $Q(z|X)$ needs to be a complex function and in order to model it we are going to use once again a neural network. This neural network has to approximate the distribution $P(z|X)$ expressed in (2.2), for this reason we are going to use the KL divergence introduced in section 2.1.3

$$\mathcal{D}_{KL}[Q(z|X)||P(z|X)] = \mathbb{E}_{z\sim Q(z|Z)}[\log Q(z|X) - \log P(z|X)] =$$
$$\mathbb{E}_{z\sim Q(z|X)}[\log Q(z|X) - \log \frac{P(X|z)P(z)}{P(X)}] =$$
$$\mathbb{E}_{z\sim Q(z|X)}[\log Q(z|X) - \log P(X|z) - \log P(z)] + \log P(X) =$$
$$\mathcal{D}_{KL}[Q(z|X)||P(z)] - \mathbb{E}_{z\sim Q(z|X)}[\log P(X|z)] + \log P(x)$$

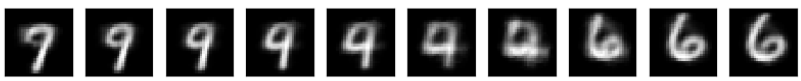


Figure 2.4: Images generated following the path $(-1,-1) \rightarrow (-1,1)$

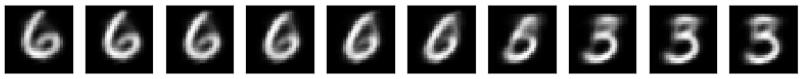


Figure 2.5: Images generated following the path $(-1,1) \rightarrow (1,1)$

Rearranging the terms:

$$\log P(X) - \mathcal{D}_{KL}[Q(z|X)||P(z|X)] = \mathbb{E}_{z\sim Q(z|X)}[\log P(X|z)] - \mathcal{D}_{KL}[Q(z|X)||P(z)] \tag{2.3}$$

The right hand side of the equation is what we want to maximize. Let's analyze the term $\mathcal{D}_{KL}[Q(z|X)||P(z)]$. We know that $P(z)$ is distributed as standard normal distribution, while we want to model $Q(z|X)$ as a normal distribution whose parameters depend on our data set X, in other words we want $Q(z|X)$ to be $\mathcal{N}(\mu(X), \Sigma(X))$ where $\mu(X)$ and $\Sigma(X)$ are neural networks.
The KL divergence between two multivariate gaussians is

$$\mathcal{D}_{KL}[\mathcal{N}(\mu_0, \Sigma_0)||\mathcal{N}(\mu_1, \Sigma_1)] = \frac{1}{2}[(\mu_0 - \mu_1)^\intercal \Sigma_1^{-1}(\mu_0 - \mu_1) + Tr(\Sigma_0\Sigma_1^{-1}) - \log\frac{|\Sigma_0|}{|\Sigma_1|} - n]$$

applied to our case

$$\mathcal{D}_{KL}[\mathcal{N}(\mu(X), \Sigma(X))||\mathcal{N}(0, I)] = \frac{1}{2}[\mu(X)^\intercal \mu(X) + Tr(\Sigma(X)) - \log|\Sigma(X)| - n]$$

as shown in appendix A.
The second addend of the left hand side of 2.3 is considered an error term, while $logP(X)$ is the log-likelihood (what we want to maximize). In pseudo code the last expression becomes

```
kl_loss = 0.5*sum(square(z_mean) + exp(z_log_var) - z_log_var - 1)
```

Where we both model z_mean and z_log_var (the log of the variance) as neural networks whose input layers is $X$, note that z_log_var is not a matrix but a vector, since it represents the diagonal of the covariance matrix (which is a diagonal matrix). We also use the fact that the determinant of a diagonal matrix is the sum of the trace of that matrix.
We still need to analyze $\mathbb{E}_{z\sim Q(z|X)}[\log P(X|z)]$ that is the log-likelihood of $P(X|z)$, which for gaussians distributions becomes the mean squared error.
The total loss function of our neural network in pseudo code becomes

```
reconstruction_loss = mse(input, output)
vae_loss = K.mean(reconstruction_loss + kl_loss)
```

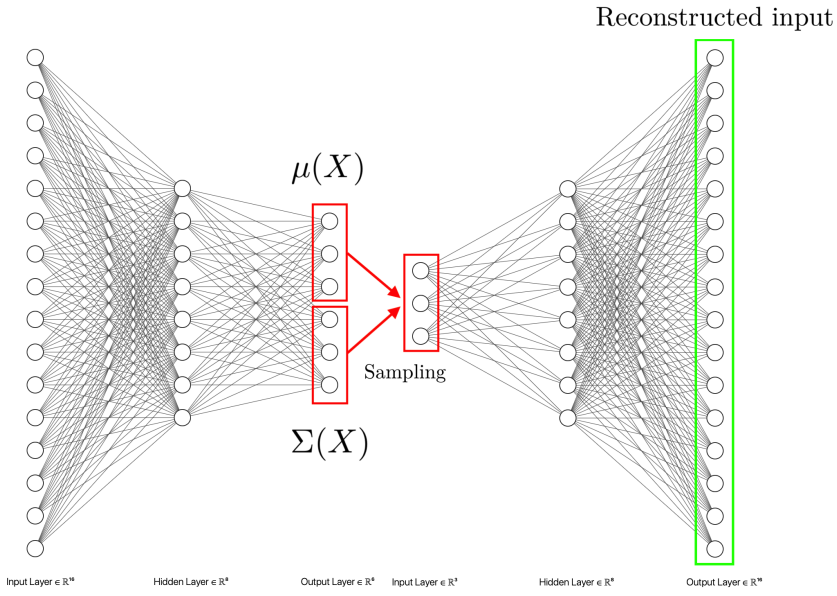



Figure 2.6: The structure of the Variational Autoencoder

## 2.3 Similiarities with classic Autoencoder

The autoencoder is a type of neural network used for non-linear dimensionality reduction, image denoising, image restoration. The main concept is to have the output layer to be of the same size and shape of the input layer while introducing a bottleneck in the architecture. In this way the autoencoder is forced to learn a lower dimension representation of our dataset. The name variational autoencoder derives from the parallelism with the classic autoencoder, since in both networks we can identify an encoder and a decoder.

### 2.3.1 Autoencoders

The structure of the autoencoder is very simple, it only consists in a feed-forward neural network where the number of output nodes is exactly the same as the number

of input nodes. Since these kind of networks are used for dimensionality reduction purposes we have to introduce a bottleneck in the structure of the feed forward network.

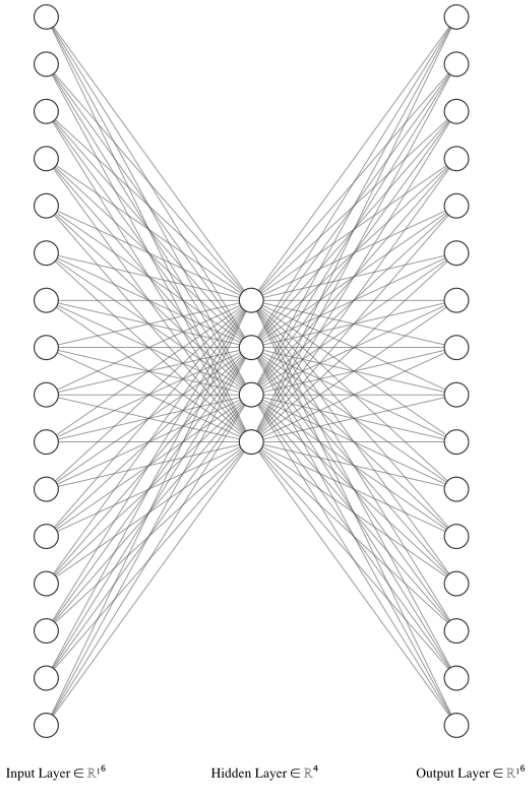



Figure 2.7: An example of a classic autoencoder

In order to ensure that the last layer of the network outputs a reconstruction of the input, during training time we minimize the reconstruction error.

$$\mathcal{L}(X;\theta) = ||X - f_\theta(g_\phi(X))||^2$$

In this way the network is forced to learn a low dimension representation of our dataset. It's worth noting that without the bottleneck in our architecture the loss function would be trivial to minimize since the network could copy the values of the input layer through the hidden layer and then passing those same values to the

output layer.
The first part of our network is the encoder, and after training can be used to output a lower dimension array that represent the initial data, while the second part is the decoder network, and after training can be used to restore the compressed data.

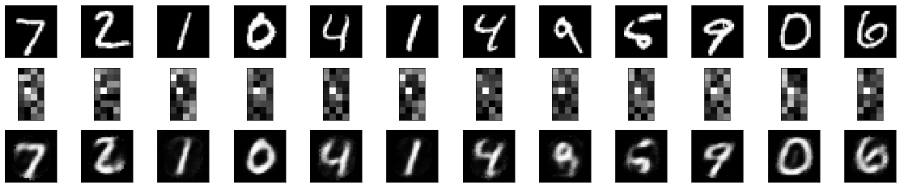


Figure 2.8: The first row shows the original data, the second row shows a representation of the compressed data, while the third row shows the reconstructed data

### 2.3.2 Autoencoders as generative models

Once the training process is completed we can detach the decoder from the network and use it to generate data, however the classic autoencoder is not able to generate digits as we can see from the examples in figure 2.9. one of the reason why is that we never constrained the "latent" representation $z$ to lay in a specific region of the vector space, which means that sampling from a normal distribution $\mathcal{N}(0, I)$ is completely arbitrary, as it would be arbitrary to sample from any other distribution.

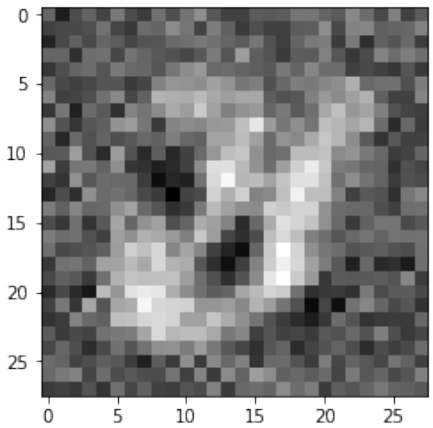



Figure 2.9: Output of the decoder whose input was a sample from a isotropic multivariate gaussian

Even perturbing the compressed version of a digit in our dataset and then reconstructing it does not provide any useful insight on how the autoencoder can be used as a generative model. It does not change any of the characteristic of the original digit in a useful way (for example changing the thickness of the stroke or transitioning to another digit) as shown in figure 2.13.

Figure 2.10: Original image next to the perturbed images.

### 2.3.3 Autoencoders an anomaly detectors

Even though the autoencoder doesn't model a probability distribution, neither in the latent space nor in the original space, we can still use it to perform anomaly detection. The anomaly score is the mean square error between the input and the reconstructed data, in this way we are assuming that the autoencoder does a better job reconstructing samples from a distribution it encountered at training time than samples it never saw before. Since the autoencoder cannot get different samples in the latent space the reconstruction error used as anomaly score is a deterministic measure, while the anomaly score used in the VAE is a probabilistic one. The metric used in order to evaluate the models is described in chapter 3. Results in the next table.

## 2.4 Plotting latent varibales in 2D

We will now try to further understand the differences between the variational autoencoder and the classic autoencoder.
In order to do this we are going to use an autoencoder whose bottleneck has dimension 2, and a variational autoencoder whose latent variable has dimension equal to 2 (which means the mean and variance used to sample data have dimension 2).
After training both models with training data we will plot the embedded version of the test data. For the VAE we are going to plot the conditioned mean in $z$ for each

| Anomaly Digit | Autoencoder AUC | Convolutional Autoencoder AUC |
|---|---|---|
| 0 | 0.67 | 0.85 |
| 1 | 0.12 | 0.32 |
| 2 | 0.78 | 0.97 |
| 3 | 0.57 | 0.94 |
| 4 | 0.48 | 0.92 |
| 5 | 0.64 | 0.95 |
| 6 | 0.72 | 0.91 |
| 7 | 0.61 | 0.71 |
| 8 | 0.57 | 0.95 |
| 9 | 0.51 | 0.69 |

Table 2.1: Area under the ROC curve for each digit for autoencoders.

sample. Looking at the plot of the variational autoencoder we can immediately note that points representing the same digit are clustered together, but we can also note that the whole complex is disposed in bivariate gaussian fashion (which is what we were trying to accomplish when building our loss function).
On the other hand in the plot of the classic autoencoder points are not particularly clustered and they don't form any particular shape.

## 2.5 Detecting anomalies with VAE

Once we built our VAE we need to train it using nominal data from the MNIST dataset (a dataset formed by hand written digits, more details in the next chapter). Since MNIST does not have explicit labels for anomalies detection purposes we are going to define anomalous and nominal data ourselves by removing one class from the entire dataset and labeling as anomalous. All the other classes will passed to the neural network at training time, in this way the variational autoencoder will learn to model those classes.
The next step is building a metric that will be used at test time in order to identify anomalous data. In order to do that we will feed a test example to the encoder part of our network, which will output its mean and variance in the latent space. We will then sample from a multivariate normal distribution whose mean and variance are those output by the encoder (remember that the produced variance is a vector that represents the diagonal of a diagonal matrix). After that we will feed the sampled $z$

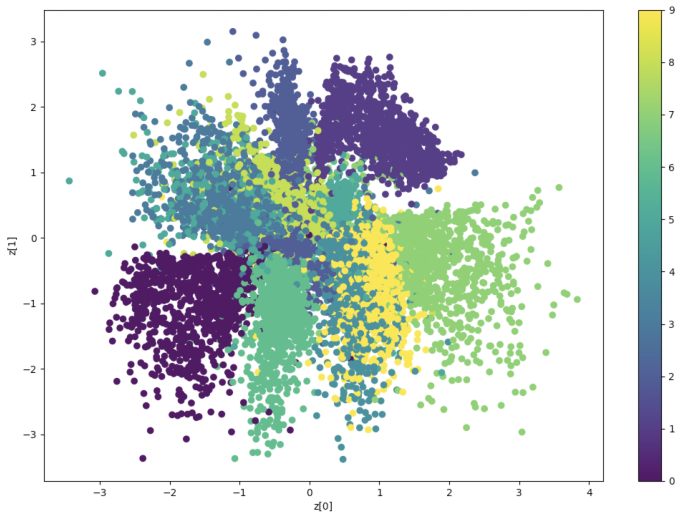



Figure 2.11: 2D plot of (variationally)autoencoded digits.

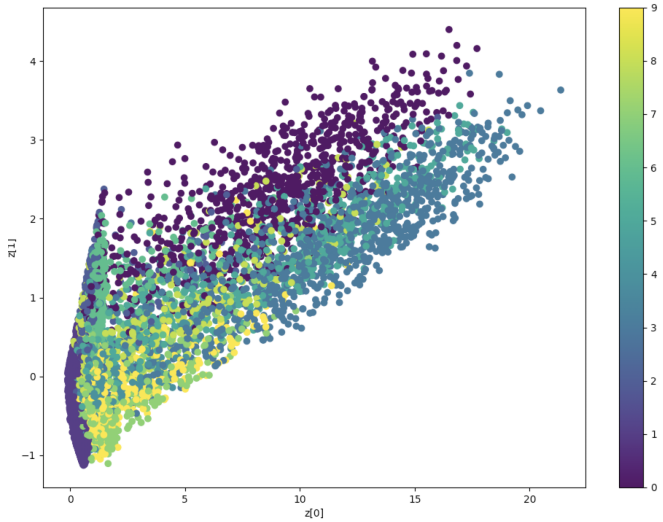



Figure 2.12: 2D plot of autoencoded digits.

in the latent space to our decoder network, that will produce a reconstructed version of the initial input data. If the the input example belongs to one of digit classes that the autoencoder encountered during training the reconstruction should be relatively accurate, while if it belongs to the anomalous class the reconstruction should not resemble the initial input.

For example when when the VAE has been given every digit except "4" as training

data it fails to reconstruct images that represent the number 4 properly. It's worth noting that the VAE thinks the input image is a 9 and reconstruct it as such.

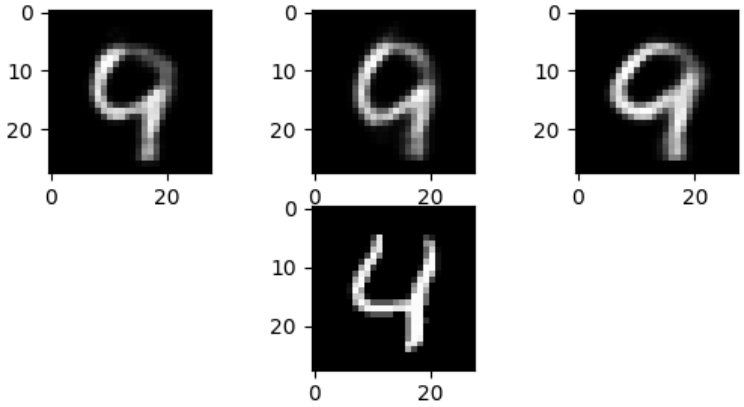



Figure 2.13: Input image "4" in the bottom row. Three samples of reconstructions of that very "4" in the top row.

The actual anomaly score in constructed similarly to the one used in classic autoencoder: the squared difference between the original input and the reconstructed one is computed, but this time it's averaged over the number of samples we decided to take from the normal distribution in the latent space $z$.

$$score = \sum_{l=1}^{L} (\hat{X}_l - X)^2$$

Where $L$ is the number of samples, $\hat{X}_l$ is the reconstruction of the $l^{th}$ sample, and $X$ is the original input image. One of the advantages of using a variational autoncoder instead of a classic one is that it's possible to get more than one sample from the distribution in the $z$ vector space, thus getting more than one reconstruction of the original image. In my experiments I didn't find the increasing of $L$ to give substantially better results.

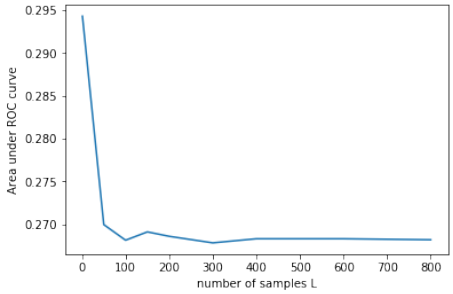



Figure 2.14: AUC when the number of samples varies for digit 1

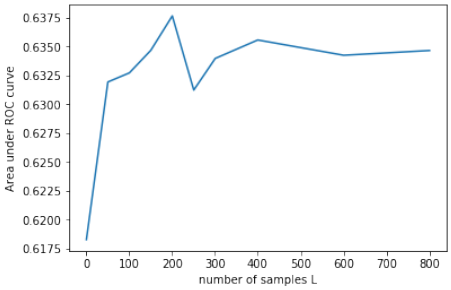



Figure 2.15: AUC when the number of samples varies for digit 9

# Chapter 3

# Experiments

## 3.1 Histograms, ROC curves, Area under the curve

In order to asses how and if the VAE is working we will use the anomaly score produced ad test time. We will then build two histograms of said score, one over anomalous data, one over normal data, in order to visualize if anomalous data has an higher reconstruction error.
While these histograms provide a visual aid of how the VAE is operating it does not give us a number that we can use to compare different results, for this reason we need to introduce the Receiver Operating Characteristic (ROC) curve, and the Area Under the Curve (AUC).

### 3.1.1 ROC

Given two overlapping gaussians distributions (a positive class and a negative class) our task is to understand how separable are those two classes. For this reason we need to choose a threshold and see how much area of the gaussian curve lays in the wrong side of the threshold for each of the two classes.

Since we will not be dealing with the explicit densities of the distributions but from the respective histograms instead of looking at the area of the density we will use the True Positive Rate (TPR) and the False Positive Rate (FPR).

$$TPR = \frac{TP}{TP + FN}$$

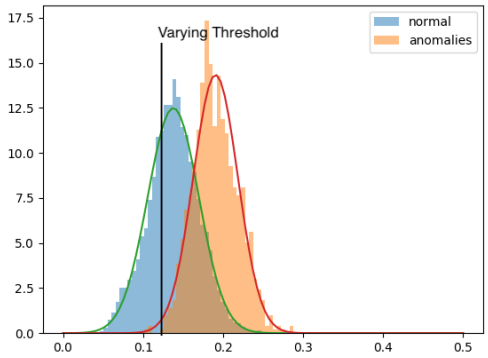



Figure 3.1: Example of the histograms for two gaussians

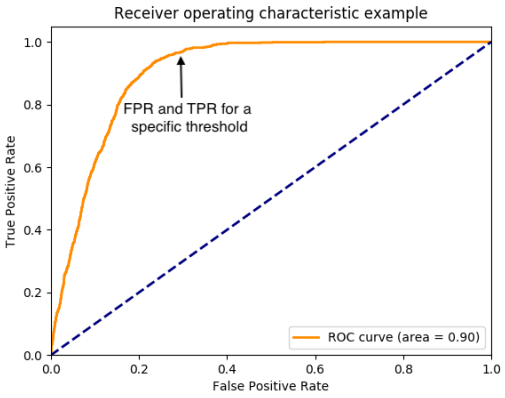



Figure 3.2: Example of a ROC curve

$$FPR = \frac{FP}{FP + TN}$$

Where $TP$ (True Positive) is the number of positive examples classified correctly, $FN$ (False Negative) is the number of examples whose prediction was negative but whose true condition was positive, $FP$ (False Positive) is the number of examples whose prediction was positive but whose true condition was negative, while $TN$ (True Negative) is the number of negative examples classified correctly. More specifically the ROC curve plots the True Positive Rate and the False Positive Rate on a plane, while the deciding thresholds varies. This means that every point of the curve corresponds to a specific threshold value, but we are not interested on the value per se, making it a useful tool for comparing models where the thresholds are

in different scales.
If the two gaussians distributions were fully separable the curve would form a $90^o$ angle, since every point corresponding to thresholds over the left (negative) gaussian would have TPR of 1, while every point corresponding to thresholds over the right (positive) gaussian would have FPR od 1,thresholds the manage to completely separate the two gaussians would have TPR of 1 and FPR of 0. On the other hand if the two gaussians had the exact mean and variance (meaning that they are not separable in any way) the ROC curve would lay on the bisector of the plane.

## 3.2 MNIST results

A series of experiments have been performed to evaluate efficacy of the variational autoencoder as an anomaly detection algorithm. The first dataset we used is MNIST [13], which contains images of handwritten digits, ranging from 0 to 9. In order to use it for anomaly detection we labelled only one digit has anomalous data and every other digit has normal digit, and performed the experiments for every possible digit as anomalous.
There are 60000 images in the training set by default (6000 for each digit), but since we removed one there are 54000 images in our custom training data. There are 10000 images in the test set, 1000 of which are labeled as anomalies.
The training is done only with normal data in an unsupervised fashion, in this way the model will learn to generate only the classes we put in the normal dataset, and will hopefully have trouble reconstructing anomalies, since it wasn't presented at training time.

It is now clear that a good numeric indicator of the performance of the model would be to compute the area under the ROC curve, which will be close to 1 for very good models that can separate classes, while would be close or under 0.5 for model that are not able to separate the two classes.

We tested to different architectures, details are presented in the next table.
In the third layer there are both the mean and the variance of the VAE model, each with 200 nodes.

The architecture labeled Dense NN1 is the one used by Jinwon An and Sungzoon Cho in "Variational Autoencoder based Anomaly Detection using Reconstruction

| Dense Architectures | | | | | |
|---|---|---|---|---|---|
| Dense NN 1 | | | Dense NN 2 | | |
| Type | #Nodes | Activation | Type | #Nodes | Activation |
| Input | 784 | | Input | 784 | |
| Dense | 400 | relu | Dense | 400 | relu |
| Dense* | 200 - 200 | linear | Dense* | 400 - 400 | linear |
| Sample | 200 | | Sample | 400 | |
| Dense | 400 | relu | Dense | 400 | relu |
| Output | 784 | relu | Output | 400 | relu |

Table 3.1: Architectures of the dense neural networks.

| Convolutional architecture | | | | |
|---|---|---|---|---|
| Type | Filters/Nodes | Stride | Activation | Outputshape |
| Input | | | | $28 \times 28 \times 1$ |
| Conv | $3 \times 3 \times 32$ | 2 | relu | $14 \times 14 \times 32$ |
| Conv | $3 \times 3 \times 64$ | 2 | relu | $7 \times 7 \times 64$ |
| Flatten | | | | 3136 |
| Dense | 200 | | relu | 200 |
| Dense* | 100 - 100 | | linear | 100 - 100 |
| Sample | | | | 100 |
| Dense | 3136 | | | 3136 |
| Reshape | | | | $7 \times 7 \times 64$ |
| Deconv | $3 \times 3 \times 64$ | 2 | relu | $14 \times 14 \times 64$ |
| Deconv | $3 \times 3 \times 32$ | 2 | relu | $28 \times 28 \times 32$ |
| Deconv | $3 \times 3 \times 1$ | 1 | sigmoid | $28 \times 28 \times 1$ |

Table 3.2: Details of the convolutional architecture.

Probability" [1]. The main differences between NN1 and NN2 is that in NN2 the data is projected in a vector space with a bigger dimension with respect to NN1.

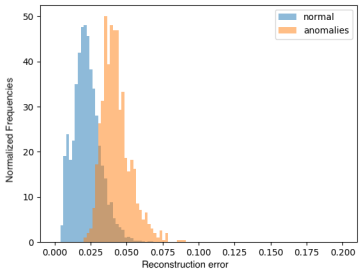



Figure 3.3: Histogram of anomalies score, 0 vs all

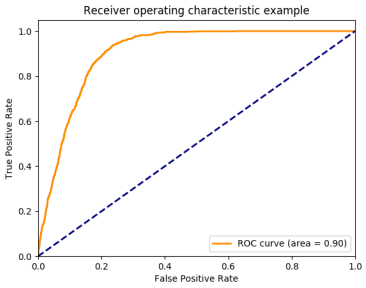



Figure 3.4: ROC curve of 0

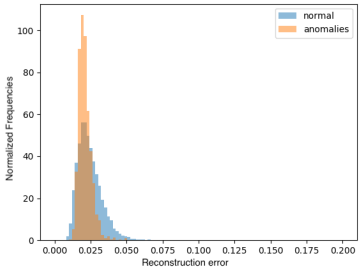



Figure 3.5: Histogram of anomalies score, 1 vs all

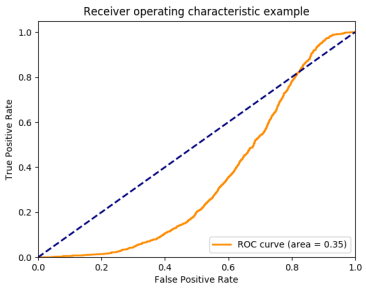



Figure 3.6: ROC curve of 1 vs all

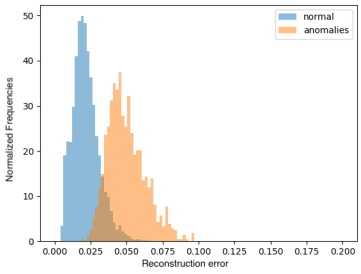



Figure 3.7: Histogram of anomalies score, 2 vs all

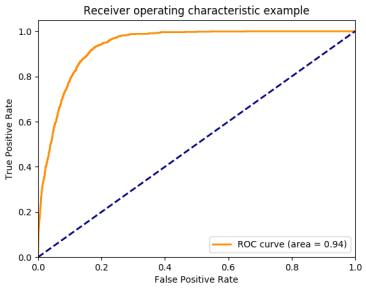



Figure 3.8: ROC curve of 2 vs all

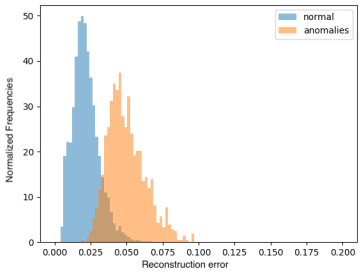



Figure 3.9: Histogram of anomalies score, 3 vs all

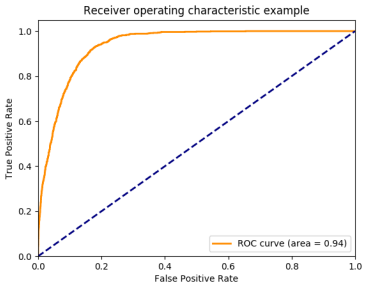



Figure 3.10: ROC curve of 3 vs all

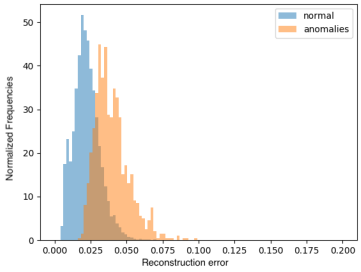



Figure 3.11: Histogram of anomalies score, 4 vs all

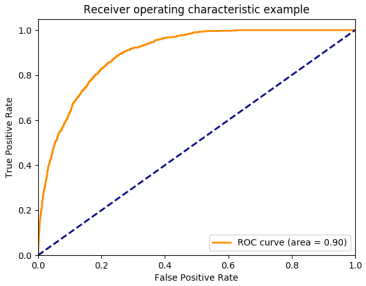



Figure 3.12: ROC curve of 4 vs all

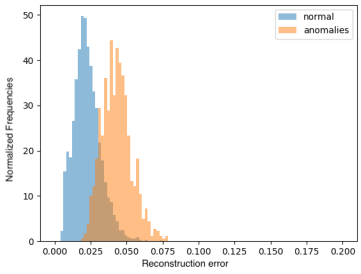



Figure 3.13: Histogram of anomalies score, 5 vs all

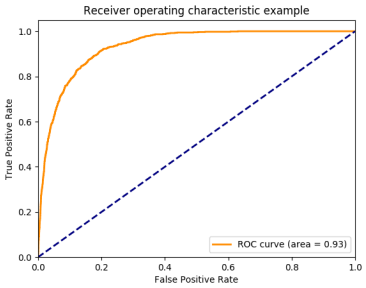



Figure 3.14: ROC curve of 5 vs all

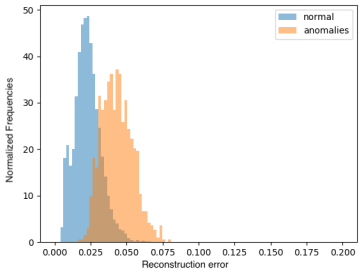



Figure 3.15: Histogram of anomalies score, 6 vs all

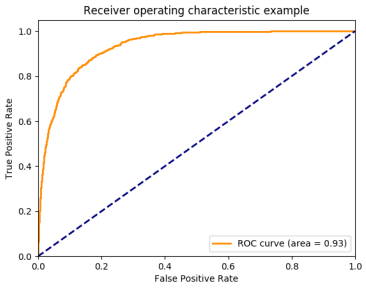



Figure 3.16: ROC curve of 6 vs all

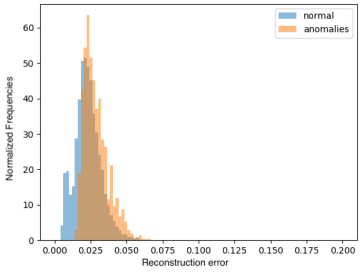



Figure 3.17: Histogram of anomalies score, 7 vs all

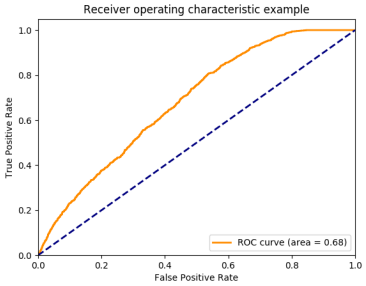



Figure 3.18: ROC curve of 7 vs all

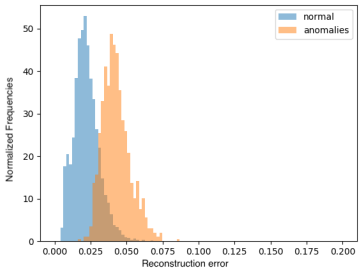



Figure 3.19: Histogram of anomalies score, 8 vs all

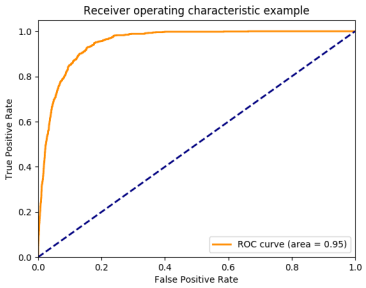



Figure 3.20: ROC curve of 8 vs all

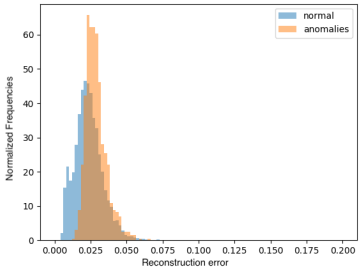



Figure 3.21: Histogram of anomalies score, 9 vs all

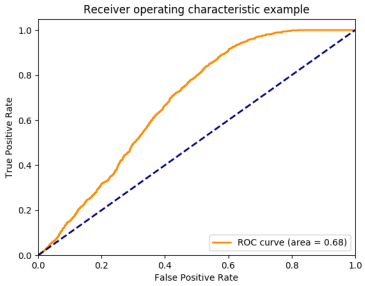



Figure 3.22: ROC curve of 9 vs all

The digits where the VAE performs the worst are 1,7 and 9. It looks like the variational autoencoder manages to reconstruct both 1 and 7 since they are both very simple symbols that the autoencoder leanrs to draw even if it never saw 1s and 7s at training time. This hypothesis is strengthened by the fact that the performance gets worst for those 2 digits when a convolutional (variational autoencoder) is used: convolutional neural networks are able to learn more complex strokes, and a simple vertical dash can be easily learned by other digits.
In particular when 1 is the anomalous digit the VAE reconstruct it as a very thin 8, and the reconstruction error between those two is very small. On the other hand it

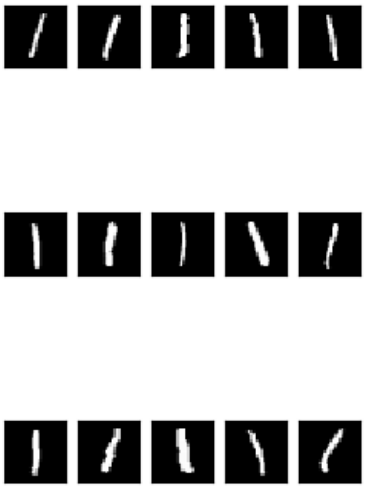


Figure 3.23: Original 1s

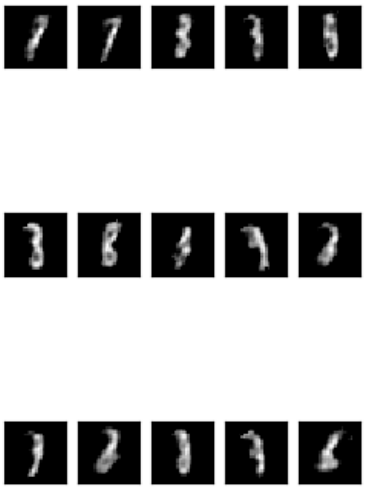


Figure 3.24: Reconstructed 1s (by the convolutional VAE)

looks like 9 under performs because it's very similar to 8, and the reconstruction error between 8s and 9s is very small. Another interesting thing to note is the shape of the ROC curve for the handwritten digit 1, since it changes convexity at right of the curve. This kind of behaviour cannot happen if the histograms where representative of to gaussians.

When the deciding threshold is at around 0.030 there are more normal examples to the right of the threshold (hence being misclassified) than anomalous ones. This is reflected in the ROC curve: there is an area in which the curve is under the bisector, But when the deciding threshold is at the beginning of the two histograms the are more anomalous examples to the right of the deciding threshold.

| Anomaly Digit | VAE NN2 | CONV VAE | AE | Jinwon VAE [1] | PCA |
|---|---|---|---|---|---|
| 0 | 0.90 | 0.85 | 0.67 | 0.91 | 0.71 |
| 1 | 0.35 | 0.32 | 0.12 | 0.14 | 0.15 |
| 2 | 0.94 | 0.97 | 0.78 | 0.92 | 0.85 |
| 3 | 0.88 | 0.94 | 0.57 | 0.78 | 0.60 |
| 4 | 0.90 | 0.92 | 0.48 | 0.81 | 0.55 |
| 5 | 0.93 | 0.95 | 0.64 | 0.86 | 0.70 |
| 6 | 0.93 | 0.91 | 0.72 | 0.85 | 0.84 |
| 7 | 0.68 | 0.71 | 0.61 | 0.60 | 0.70 |
| 8 | 0.95 | 0.95 | 0.57 | 0.89 | 0.61 |
| 9 | 0.68 | 0.69 | 0.51 | 0.54 | 0.47 |

Table 3.3: Area under the ROC curve for each digit and for each model tested.

## 3.3 Adding impurities to the training data

In the real world is very hard to gather completely clean data, and hand labeling can be expensive and time consuming. For this reason we are going to test how the model performs when we give a small percentage of anomalous data at training time. We tested 1% and 0.1%.

While every digit performed significantly worst with 1% of impurities we have to take into account the number of different classes the VAE is learning to model only in the normal data. There are 9 classes only in the normal data, which means the impurities are actually close to 10% in number with respect to another normal class.

Both with 1% and 0.1% the class 1 is the one that performed the worst, which means that our model does not need a big number of samples in order to model a simple vertical dash.

| Anomaly Digit | VAE | 1% Impurities | 0.1% Imputirites |
|---|---|---|---|
| 0 | 0.90 | 0.77 | 0.89 |
| 1 | 0.35 | 0.09 | 0.24 |
| 2 | 0.94 | 0.92 | 0.95 |
| 3 | 0.88 | 0.79 | 0.83 |
| 4 | 0.90 | 0.78 | 0.84 |
| 5 | 0.93 | 0.87 | 0.91 |
| 6 | 0.93 | 0.78 | 0.90 |
| 7 | 0.68 | 0.53 | 0.71 |
| 8 | 0.95 | 0.88 | 0.92 |
| 9 | 0.68 | 0.57 | 0.60 |

Table 3.4: Area under the ROC curve for VAE when adding impurities.

## 3.4 A note on one class SVM

We tested one class SVM with the exact same set up as the for the VAE but from our experiments it seemed that the model had troubles modeling the normal class, since it's formed by multiple subclasses.
For this reason we tested one class SVM with only one digit as normal class and the other digits as outliers (the opposite of what we have done previously). With this configuration the one class SVM with $rbf$ kernel is able to recognize outliers.
It should be noted that the one class SVM does not output a score, but it predicts

| Normal Digit | One class SVM |
|---|---|
| 0 | 0.68 |
| 1 | 0.96 |
| 2 | 0.58 |
| 3 | 0.67 |
| 4 | 0.72 |
| 5 | 0.64 |
| 6 | 0.70 |
| 7 | 0.79 |
| 8 | 0.63 |
| 9 | 0.79 |

Table 3.5: Area under the ROC curve for the one class SVM

directly outliers and normal examples at testing time; this means that the ROC curve consists in only one point, with coordinates true positive rate and false positive rate.

All digits have TPR of 1 and varying FPR (which determines the AUC).
The one class SVM is able to model the digit 1,7 and 9 fairly well, and while this seems in contrast to what happened with the VAE and AE we have to take into account the fact that we are in the opposite scenario as before. We can then interpret this result by saying that even for the one class SVM those are the easiest classes to model, which is consisten with what we saw in the previous sections.

## 3.5 KDDCUP99 results

The original KDD99 dataset is a dataset widely used as a benchmark in fraud detection. Each entry of the dataset describes a computer network connections, it has 41 attributes (of which 34 are continuous and 7 categorical). We are using a modified version with only 4 attributes: service, duration, src_bytes, dst_bytes. The only categorical attribute is service with 5 categories, http, smtp, ftp, ftp_data, others. Only entries with service=http are used here.
There are 3,377 entries that are "attacks" (anomalies in our context) of the 567,497 total entries.
The architecture of both the encoder and the decoder is very simple. The input layer has 3 nodes, the only one hidden layer modeling the latent space has 2 nodes, and the output node has again 3 nodes.

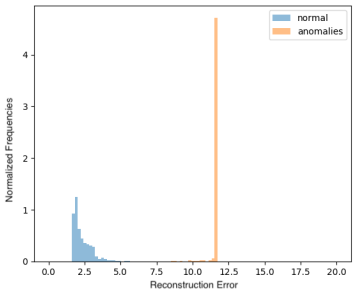



Figure 3.25: Histogram of anomalies score for the http dataset (VAE).

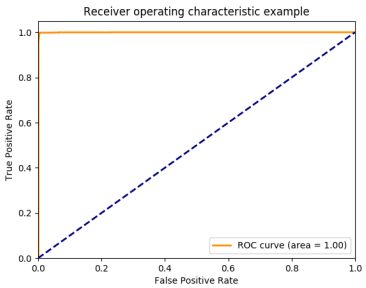



Figure 3.26: ROC curve for the http dataset (VAE).

As you can see from Figure 3.31 and 3.32 the model manages to separate completely attacks and regular network connections. The ROC curve is made of only

one point with coordinates (0,1) since the histograms are completely separable. The one class SVM manages to separate anomalies almost perfectly, as the VAE manged to do. The isolation forest is also able to reach the same level of performance of the one class SVM and the VAE, with a significant shorter training time.

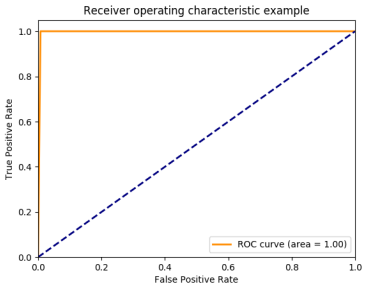



Figure 3.27: ROC curve for the http dataset using the one class SVM

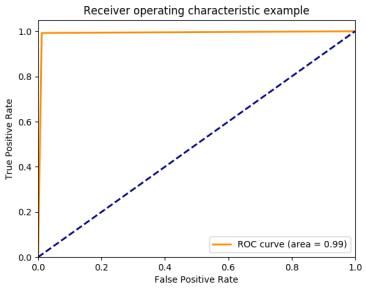



Figure 3.28: ROC curve for the http dataset using the isolation forest

A similar experiment has been performed but using entries with service=smtp. More information about both dataset can be found at [16] and [8]

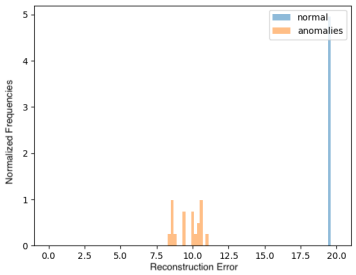



Figure 3.29: Histogram of anomalies score for the smpt dataset (VAE).

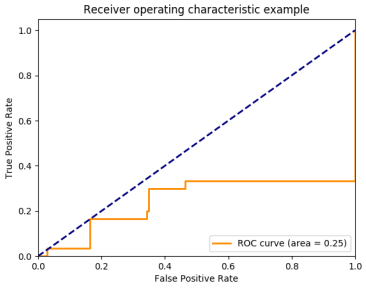



Figure 3.30: ROC curve for the smtp dataset (VAE).

While from the histogram it looks like the two classes are separated the anomalies have smaller reconstruction error, which is not the intended behaviour. This is reflected by the fact that the ROC curve lies under the diagonal, which suggests that our model is performing worse than a random choice. Given the results of the MNIST experiments we could speculate and say that the attacks in the smtp dataset have a less complex structure that the VAE is able to learn, but this would need

further assessments.

While the one class SVM manages to separate the two classes, the isolation forest fails completely since the ROC curve lies on the diagonal.

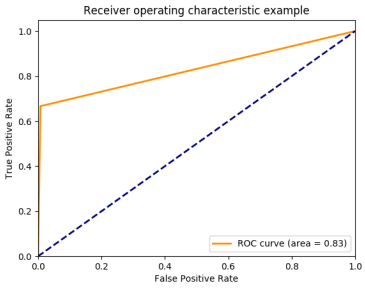



Figure 3.31: ROC curve for the smtp dataset using the one class SVM

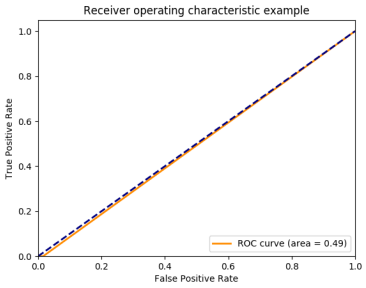



Figure 3.32: ROC curve for the smtp dataset using the isolation forest

# Chapter 4

# Language processing

## 4.1 Problem setup

We will now try to apply the techniques used in previous chapters in natural language processing.
The ideal scenario for this application would be monitoring a community that writes text with a common interest (for example an online forum) and understand when the members of this community would start talking about a new topic.

## 4.2 Dataset and preprocessing

In order to simulate this situation the Sentence Classification Dataset /citesentence has been used. While this dataset provides labels for a small subset of the data, we used the much larger unlabeled portion, since we will be using unsupervied learning techniques.
The data set consists of a corpus made of academics articles, each belonging to one of these three fields:

- Computational Biology
- The machine learning repository on Arxiv
- The psychology journal Judgment and Decision Making

Two of these fields will be labeled as normal data while the other one will be labeled as anomalous data. Each article will be split in sentences, such that one sentence will be one row of our dataset.

## 4.3 Universal Sentence Encoder

Now that we have a dataset made of different sentences from different fields we need to transform it in something the our VAE can process, namely an array of numbers. This process is usually called feature extraction
In order to decrease the number of unique words we applied a common algorithm in natural language processing called stemming. Stemming consists in identifying common prefixes in different words so that words like "playing", "plays", "played" were all mapped to "play". The next step consisted in eliminating from the sentences the words that appeared too often (usually called stopwords), since those are probably not field specific, and eliminating the words that appeared too sporadically.
At this point we have to actually create an array of numbers, the approach we used is called Bag of Words and it works like this: An array that has as dimension the number of unique words in our dictionary with 0 in all of its entries is created. For each sentence one element of that array corresponds to the counter of how many times that word appears in that sentence (this process is usually called CountVectorizer). It's clear that even after stemming and removing stopwords the size of the vocabulary will still be very big, which means that vectors will be long and most importantly very sparse.
After these vectors are created they are fed to our VAE like we did for MNIST images. Our model was not able to recognize anomalous sentences, my hypothesis is that this happened since my dataset consists only of 36167 sentences, not enough for the VAE to learn how to represent them given that the vectors are very sparse. We need to find a way to create a numeric array that represents our sentence that is not sparse, but since our dataset is still relatively small we are going to use the concept of transfer learning.
Transfer learning is a technique that consists in leveraging pretrained models in order to train our model on top of the pretrained one. More specifically we are going to use the Universal Sentence Encoder [6]. This tensorflow model takes as input words, sentences, or full paragraphs and gives as output a 512 array that represents what

was gives as input. This means that sentences with similar meaning such as "How are you today?" and "How do you feel today" are going to be mapped as adjacent vectors in the new space.
It is possible to compute a semantic textual similarity between to sentences by using the inner product of the to respective 512 arrays.

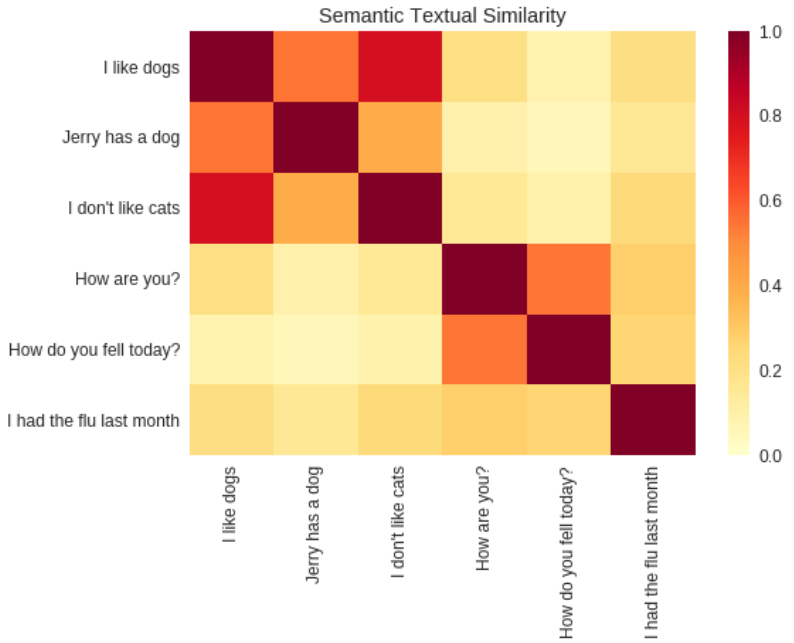



Figure 4.1: Semantic textual similarity of simple sentences

Semantic similarity between short sentences gives the desirable result, while it has trouble with long sequences.

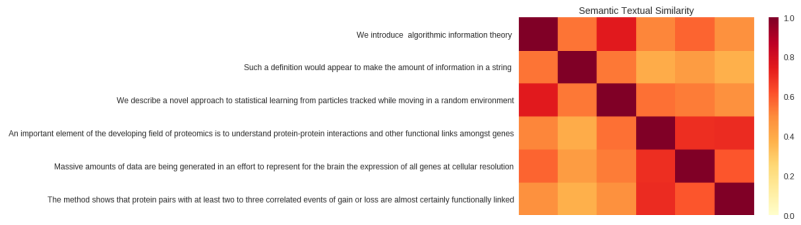



Figure 4.2: Semantic textual similarity: the first three sentences are from the Arxiv papers, while the second three are from the biology journal

## 4.4 VAE application and results

We both gave as input of the Universal Sentence Encoder full sentences without any preprocessing and sentences preprocessed with our stemming and CountVectorizer. Sentence without preprocessing performed considerably better.
Class 1 corresponds to sentences from the Arxiv machine learning repository, class 2 corresponds to sentences from the psychology journal, while class 3 corresponds to sentences from the computational biology journal.

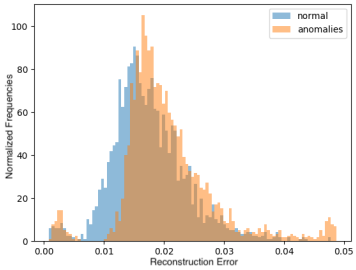



Figure 4.3: Histogram of reconstruction error when class 1 is anomalous

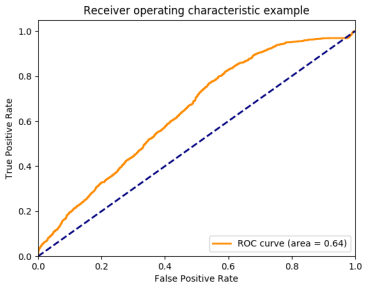



Figure 4.4: ROC curve when class 1 is anomalous

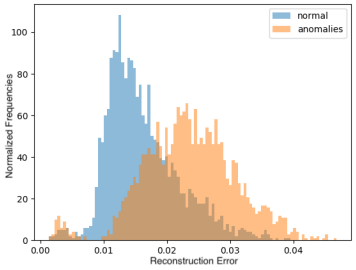



Figure 4.5: Histogram of reconstruction error when class 2 is anomalous

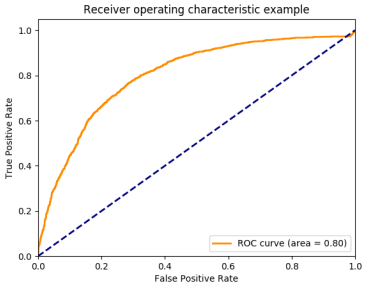



Figure 4.6: ROC curve when class 2 is anomalous

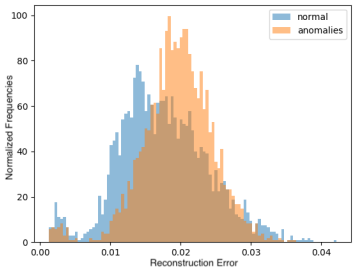



Figure 4.7: Histogram of reconstruction error when class 3 is anomalous

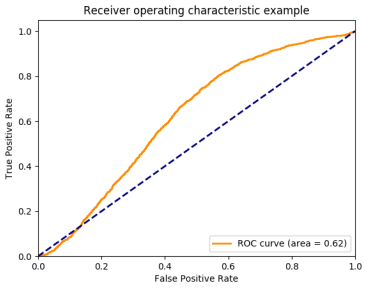



Figure 4.8: ROC curve when class 3 is anomalous

Results with stemmed, countVectorized and then encoded sentences:

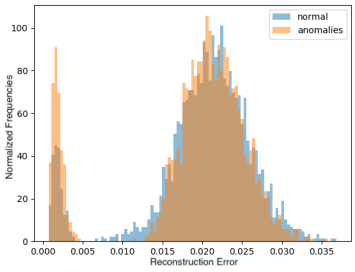



Figure 4.9: Histogram of reconstruction error when class 1 is anomalous

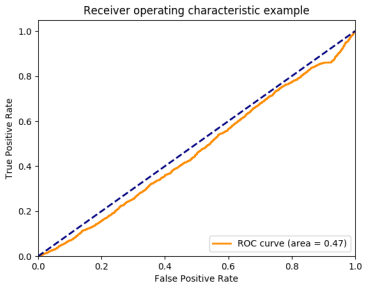



Figure 4.10: ROC curve when class 1 is anomalous

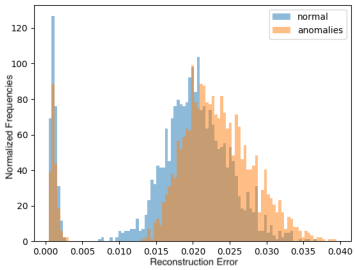



Figure 4.11: Histogram of reconstruction error when class 2 is anomalous

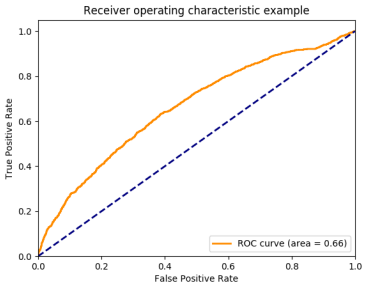



Figure 4.12: ROC curve when class 2 is anomalous

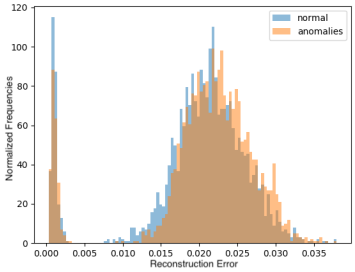



Figure 4.13: Histogram of reconstruction error when class 3 is anomalous

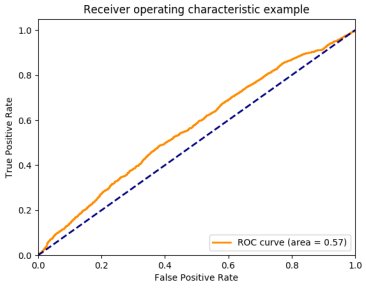



Figure 4.14: ROC curve when class 3 is anomalous

# Chapter 5

# Conclusions

From our experiments it looks like the variational autoencoder is able to solve different kinds of anomaly detection problems, from images, semantic analysis and structured Data.
It is able to deal with datasets in which the normal class is formed of multiple subclasses, each with its own characteristics, but it has problems when the anomalous class is substantially easier to model that the other ones, since it can learn to reproduce it looking at the other classes.
It has all the benefits and drawbacks of other deep learning techniques, for example it is able to take advantage of more layers provided that the amount of data used for training is appropriate. The number of hyperparameters of the architecture is substantial and many experiments are needed in order to determine the architecture for each problem.
When given a large training set it has the advantage of performing feature extraction in the first layers of the network, which means it can accept as input data with many dimension like images (where the support of the images distribution is very small considering the whole space), where methods that leverage simple statistical properties fail. Future work could focus on exploring other generative methods that model the probability distribution in latent space in an explicit way for anomaly detection purposes, like MADE [9] and NADE [18] networks.

# Appendix A

# An appendix

## A.1 KL Divergence between two multivariate Gaussians

$$X \sim \mathcal{N}(\mu_0,\, \Sigma_0)\,, \text{ with } \mu_0 \in \mathcal{R}^d, \Sigma_0 \in \mathcal{R}^{d\times d}$$
$$Y \sim \mathcal{N}(\mu_1,\, \Sigma_1)\,, \text{ with } \mu_1 \in \mathcal{R}^d, \Sigma_1 \in \mathcal{R}^{d\times d}$$

Since the two random variables are multivariate normal we now that the density of $X$ is

$$p(x) = (2\pi)^{-\frac{n}{2}}|\Sigma_0|^{-\frac{1}{2}}\exp(-\frac{1}{2}(x-\mu_0)^\intercal\Sigma_0^{-1}(x-\mu_0))$$

and the density of Y is

$$q(x) = (2\pi)^{-\frac{n}{2}}|\Sigma_1|^{-\frac{1}{2}}\exp(-\frac{1}{2}(x-\mu_1)^\intercal\Sigma_1^{-1}(x-\mu_1)$$

We can now proceed to compute the divergence between the two random variables

$$\mathcal{D}_{KL}[X||Y] = \mathbb{E}_{x\sim X}[\log X - \log Y] = \int_{-\infty}^{+\infty} p(x)[\log p(x) - \log q(x)]dx =$$

$$\int_{-\infty}^{+\infty} p(x)\log p(x)dx - \int_{-\infty}^{+\infty} p(x)\log q(x)dx$$

Let's compute the first addend of the last equation

$$\int_{-\infty}^{+\infty} p(x)\log p(x)dx = \int_{-\infty}^{+\infty} p(x)\log[(2\pi)^{-\frac{n}{2}}|\Sigma_0|^{-\frac{1}{2}}\exp(-\frac{1}{2}(x-\mu_0)^\intercal\Sigma_0^{-1}(x-\mu_0))]dx =$$

$$\int_{-\infty}^{+\infty} p(x)\{\log[(2\pi)^{-\frac{n}{2}}|\Sigma_0|^{-\frac{1}{2}}] + \log[\exp(-\frac{1}{2}(x-\mu_0)^\intercal\Sigma_0^{-1}(x-\mu_0))]\}dx =$$

$$\int_{-\infty}^{+\infty} p(x)\{\log[(2\pi)^{-\frac{n}{2}}|\Sigma_0|^{-\frac{1}{2}}] - \frac{1}{2}(x-\mu_0)^\intercal\Sigma_0^{-1}(x-\mu_0)\}dx =$$

$$\mathbb{E}_{x\sim X}[\log[(2\pi)^{-\frac{n}{2}}|\Sigma_0|^{-\frac{1}{2}}]] - \frac{1}{2}\mathbb{E}_{x\sim X}[(x-\mu_0)^\intercal\Sigma_0^{-1}(x-\mu_0))] =$$

$$\log[(2\pi)^{-\frac{n}{2}}|\Sigma_0|^{-\frac{1}{2}}] - \frac{1}{2}Tr\{\Sigma_0^{-1}\Sigma_0\} = \log[(2\pi)^{-\frac{n}{2}}|\Sigma_0|^{-\frac{1}{2}}] - \frac{1}{2}n$$

Let's compute the second addend of equation using matrix properties from the Matrix Cookbook [15]

$$\int_{-\infty}^{+\infty} p(x)\log q(x)dx = \int_{-\infty}^{+\infty} p(x)\log((2\pi)^{-\frac{n}{2}}|\Sigma_1|^{-\frac{1}{2}}\exp(-\frac{1}{2}(x-\mu_1)^\intercal\Sigma_1^{-1}(x-\mu_1)) =$$

$$\int_{-\infty}^{+\infty} (p(x)\log((2\pi)^{-\frac{n}{2}}|\Sigma_1|^{-\frac{1}{2}}) - \frac{1}{2}(x-\mu_1)^\intercal\Sigma_1^{-1}(x-\mu_1)) =$$

$$\mathbb{E}_{x\sim X}[\log((2\pi)^{-\frac{n}{2}}|\Sigma_1|^{-\frac{1}{2}})] - \frac{1}{2}\mathbb{E}_{x\sim X}[(x-\mu_1)^\intercal\Sigma_1^{-1}(x-\mu_1)] =$$

$$\log((2\pi)^{-\frac{n}{2}}|\Sigma_1|^{-\frac{1}{2}}) - \frac{1}{2}(\mu_0-\mu_1)^\intercal\Sigma_1^{-1}(\mu_0-\mu_1) - \frac{1}{2}Tr(\Sigma_0\Sigma_1^{-1})$$

Putting everything together

$$\mathcal{D}_{KL}[X||Y] = \log[(2\pi)^{-\frac{n}{2}}|\Sigma_0|^{-\frac{1}{2}}] - \frac{1}{2}n - \{\log((2\pi)^{-\frac{n}{2}}|\Sigma_1|^{-\frac{1}{2}}) - \frac{1}{2}(\mu_0-\mu_1)^\intercal\Sigma_1^{-1}(\mu_0-\mu_1) - \frac{1}{2}Tr(\Sigma_0\Sigma_1^{-1})\} =$$

$$-\frac{1}{2}\log\frac{|\Sigma_0|}{|\Sigma_1|} - \frac{1}{2}n + \frac{1}{2}(\mu_0-\mu_1)^\intercal\Sigma_1^{-1}(\mu_0-\mu_1) + \frac{1}{2}Tr(\Sigma_0\Sigma_1^{-1}) =$$

$$\frac{1}{2}[(\mu_0-\mu_1)^\intercal\Sigma_1^{-1}(\mu_0-\mu_1) + Tr(\Sigma_0\Sigma_1^{-1}) - \log\frac{|\Sigma_0|}{|\Sigma_1|} - n]$$

# Bibliography


[1] Jinwon An and Sungzoon Cho. "Variational autoencoder based anomaly detection using reconstruction probability". In: (2015).

[2] Jinwon An and Sungzoon Cho. "Variational Autoencoder based Anomaly Detection using Reconstruction Probability". In: 2015.

[3] David Applebaum. *Probability and Information: An Integrated Approach*. 2nd ed. New York, NY, USA: Cambridge University Press, 2008. ISBN: 052172788X, 9780521727884.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006. ISBN: 978-0387-31073-2. URL: http://research.microsoft.com/en-us/um/people/cmbishop/prml/.

[5] Markus M. Breunig et al. "LOF: Identifying Density-based Local Outliers". In: *SIGMOD Rec.* 29.2 (May 2000), pp. 93–104. ISSN: 0163-5808. DOI: 10.1145/335191.335388. URL: http://doi.acm.org/10.1145/335191.335388.

[6] Daniel Cer et al. "Universal Sentence Encoder". In: *arXiv e-prints*, arXiv:1803.11175 (2018), arXiv:1803.11175. arXiv: 1803.11175 [cs.CL].

[7] C. Doersch. "Tutorial on Variational Autoencoders". In: *ArXiv e-prints* (June 2016). arXiv: 1606.05908 [stat.ML].

[8] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[9] Mathieu Germain et al. "MADE: Masked Autoencoder for Distribution Estimation". In: *arXiv e-prints*, arXiv:1502.03509 (2015), arXiv:1502.03509. arXiv: 1502.03509 [cs.LG].

[10] Heiko Hoffmann. "Kernel PCA for novelty detection". In: *Pattern Recognition* 40.3 (2007), pp. 863 –874. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2006.07.009. URL: http://www.sciencedirect.com/science/article/pii/S0031320306003414.

[11] Gareth James et al. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014. ISBN: 1461471370, 9781461471370.

[12] J. Kim and C. D. Scott. "Robust Kernel Density Estimation". In: *ArXiv e-prints* (July 2011). arXiv: 1107.3133 [stat.ML].

[13] Yann LeCun and Corinna Cortes. "MNIST handwritten digit database". In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[14] Fei Tony Liu, Kai Ming Ting, and Zhi hua Zhou. "Isolation Forest". In: *In ICDM '08: Proceedings of the 2008 Eighth IEEE International Conference on Data Mining. IEEE Computer Society*, pp. 413–422.

[15] Kaare Brandt Petersen et al. *The matrix cookbook*. Tech. rep. 2006.

[16] Shebuti Rayana. *ODDS Library*. 2016. URL: http://odds.cs.stonybrook.edu.

[17] Bernhard Schölkopf et al. "Support Vector Method for Novelty Detection". In: *Proceedings of the 12th International Conference on Neural Information Processing Systems*. NIPS'99. Denver, CO: MIT Press, 1999, pp. 582–588. URL: http://dl.acm.org/citation.cfm?id=3009657.3009740.

[18] Benigno Uria et al. "Neural Autoregressive Distribution Estimation". In: *arXiv e-prints*, arXiv:1605.02226 (2016), arXiv:1605.02226. arXiv: 1605.02226 [cs.LG].

[19] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. "Pixel Recurrent Neural Networks". In: *arXiv e-prints*, arXiv:1601.06759 (Jan. 2016), arXiv:1601.06759. arXiv: 1601.06759 [cs.CV].

[20] Quan Wang. "Kernel Principal Component Analysis and its Applications in Face Recognition and Active Shape Models". In: *arXiv e-prints*, arXiv:1207.3538 (2012), arXiv:1207.3538. arXiv: 1207.3538 [cs.CV].