Università degli Studi di Padova

Department of Information Engineering

Master Thesis in ICT for Internet and Multimedia

A Stochastic Approach to Anomaly Detection and Classification in Multidimensional Time Series

Supervisor
Prof. Stefano Ghidoni
Università di Padova

Master Candidate
Ludovico Frizziero

Company tutors
Dott. Michele Giusto
Dott. Daniel Felipe Vacca Manrique
Data Reply IT

Academic Year 2018/2019 - 9/12/2019

To my family. To Corinna, my sister.

Abstract

Anomaly detection, which is the identification of rare items, has always been an important applicative field, but with the increase in the availability of data and computing power, it is experiencing a boost in its demands and promises. From batch anomaly detection, often referred to as outlier detection, to an online analysis of real-time streamed data series, the problems and challenges this branch of data science has to solve are countless. Important and central in this works is to find a method to analyze video signals from CCTV cameras in search of dangers to public safety, which then become the anomaly in this context.

A framework that tries to account for many of the challenges typical of anomaly detection is therefore devised, and an implementation, able to run in realtime on mid-range edge consumer hardware (such as normal desktop PC) is provided, based on an interplay between Convolutional Neural Networks and Hidden Markov Models. The framework and its implementation also incorporate considerations typical of action recognition in videos. The system thus obtained is tested first against the theoretical results found about the interaction between the two machine learning models, and at a second stage, on the task of identifying car crashes on real videos. Some problems related to the availability of data and further work needed on the action recognition task limit the results for the second point, but the overall system shows good promises to be expanded in the future.

Sommario

Quello dell'anomaly detection, ovvero dell'identificazione di elementi rari, è sempre stato un importante campo di ricerca oltre che applicativo, ma con l'incremento di disponibilità di dati e potenza computazionale, sta sperimentando un incremento nelle richieste e promesse a cui è tenuto. Dall'anomaly detection in batch, cioè offline, spesso chiamata anche outlier detection, all'analisi online di flussi di dati in tempo reale, i problemi e le sfide che questo ramo della data science deve affrontare sono innumerevoli. Importante e centrale per questo lavoro è trovare un metodo per analizzare segnali video da telecamere a circuito chiuso, alla ricerca di potenziali problemi per la sicurezza pubblica, considerati come le anomalie in questo contesto.

Una soluzione che cerca di tenere conto di molte delle sfide tipiche dell'anomaly detection è prima ideata, poi implementata, in modo da essere in grado di venir eseguita in tempo reale su hardware di media gamma comunemente disponibile (come quello che si trova in normali desktop PC). La soluzione è basata su Reti Neurali e Modelli di Markov Nascosti. La soluzione trovata incorpora anche alcune tipiche considerazioni in ambito di riconoscimento di azioni nei video. Il sistema così ottenuto è testato prima contro i risultati teorici trovati, poi anche nel compito di trovare incidenti stradali in video reali. Tuttavia, problemi con i dati disponibili e la necessità di maggior lavoro sul riconoscimento di azioni hanno fatto sì che i risultati in questo secondo caso siano limitati, anche se il sistema mostra buone basi per continuare futuri sviluppi.

Contents

Listing of figures

Listing of tables

Listing of acronyms

AE ..... Auto Encoder neural network

AI ..... Artificial Intelligence

CCTV ..... Close Circuit TeleVision

CNN ..... Convolutional Neural Network

EM ..... Expectation Maximization

FPS ..... Frames per Seconds

HMM ..... Hidden Markov Model

HOF ..... Histogram of Oriented Features

HOG ..... Histogram of Oriented Gradients

ML ..... Machine Learning

LSTM ..... Long Short Term Memory

MAP ..... Maximum a Posteriori

NN ..... Neural Network

ReLU ..... Rectified Linear Unit

1 Introduction

ANOMALY DETECTION is among the main applications of machine learning. Its main concern is detecting data points that *do not fit well* with the rest, and maybe also the reason why that is the case. The applications are countless and mainly related either to noise removal and data cleanup in some instances, or security matters in others, like fault detection, fraud detection, predictive maintenance, social security.

The major problem though lies in the definition itself: the concept of "not fitting well" is never clearly defined by its own nature. For a generic data point, this condition, for instance, is affected heavily by seasonal trends, which a ro-

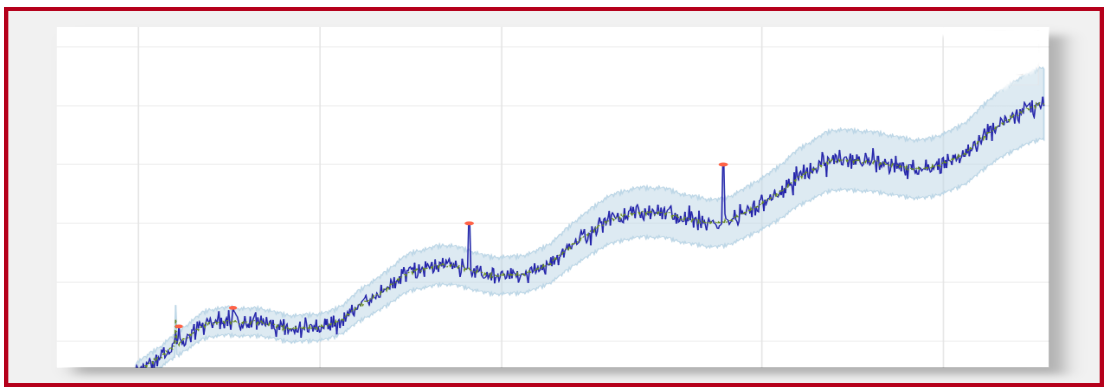


Figure 1.1: An example of a signal with anomalous samples.

bust detection system must account for, or by the context. Imagine a sudden spike in taxi rides: it may not be considered an anomaly of the signal in the context of a thunderstorm going on at the same time unless it is clearly different from other spikes that happened during thunderstorms. Other challenges an anomaly detector must face are: (i) minimizing false positives, to avoid desensitization of the human controller; this is most important especially because such systems are most often used as an aid to controllers rather than a standalone solution; (ii) generalization, that is the ability to scale to multiple types of signals, provided in large quantity at the same time as well; (iii) robustness, to avoid starting misinterpreting anomalies as common behavior, especially when continuous feedback is involved.

1.1 Common methods

There are several established methods to perform anomaly detection. The most simple one is certainly to put static thresholds on the samples (upper and lower bounds as necessary). As soon as a sample exceeds such values, it is flagged as anomalous. This, albeit straightforward and actually closely adherent with the definition of anomaly, doesn't perform well in most cases, in particular when the signal starts to become complex, both in terms of behavior, by including seasonality and contextual changes, and growing in dimensions, but also in its content (a one dimensional static signal out of a thermostat compared to a three-dimensional one, related to time, like a video for example).

1.1.1 Static methods

Many anomaly detection systems do not need to analyze time series, but rather work on static distributions [4]. Common in such cases is to revert to unsupervised clustering techniques, often based on learning a distribution to determine the best clustering arrangement. In this instance, it is more common to refer to the anomalous sample as an outlier, and many applications in this matter are related to fraud detection, networks performance analysis,

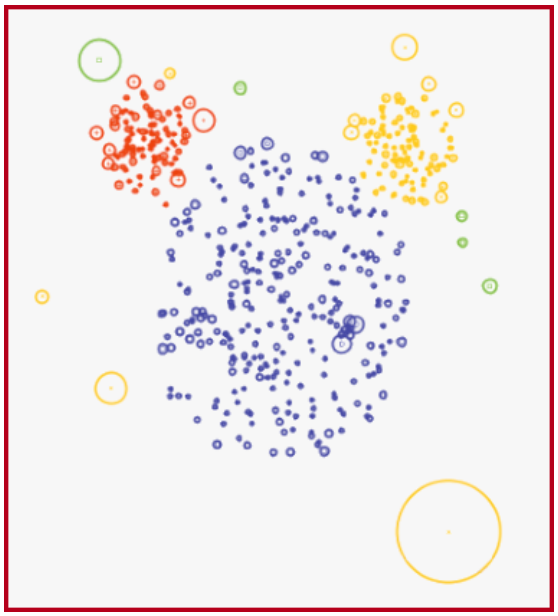


Figure 1.2: An example of a static outlier detection. The bigger the circle, the more the sample is believed to be anomalous.

intrusion detection and such. Other techniques may try instead to model either both abnormality and normality at the same time, or normality only, to later find anomalous samples through some kind of comparison metric between a predicted normal sample and the given, real, sample.

Also, statistical approaches are common. They involve modeling the data by some proper distribution, to later apply thresholds on the likelihood of new samples, or their significance level (quantiles), to understand whether or not to flag them.

A little bit more involved are graph-based methods, applicable when samples present a relational structure among each other, a condition not always true. These techniques revolve around algorithms that are able to find communities of, or rank, nodes, with perhaps one of the most famous being PageRank [5], that scores nodes based on random walks, and was behind Google's first search engine.

Lastly, deep learning methods based on artificial neural networks are also useful in this instance. In particular [6] uses a very recent and powerful architecture that goes under the name of Generative Adversarial Networks to make an algorithm learn the normal data's distribution very precisely, and a

mapping from a uniform distribution to this one, to then discover the anomalies by reconstruction error: a sample from the uniform distribution is used to generate a normal synthetic sample from the data distribution; later this is compared to the actual real sample to see how much different they are.

1.1.2 Dynamic methods

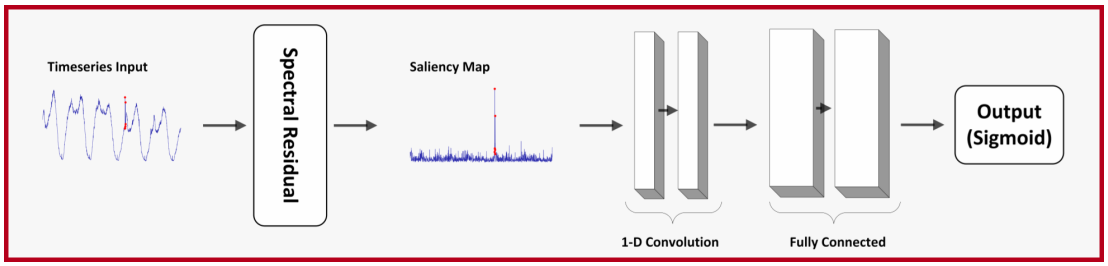


Figure 1.3: System implemented by [1].

Much more difficult is to find anomalies in time series. Aside from adopting statistical thresholds, learned by fitting some distribution to the signal's samples, or by adopting dynamically changing versions of them, one can try to first translate the signal in something simpler to manage, by some kind of method. In this new representation, it may be the case that thresholds do actually start to perform better. For instance [1] states that anomalies detection in mono-variate signals is similar to visual saliency. Hence, they employ a Spectral Residual technique [7] coupled to simple convolutional neural networks to enhance anomalies, in order to later use dynamic thresholds on the output. This results in the system of Figure 1.3. On a side note, such a system is also implemented to scale to millions of time series per minute.

But what if seasonality and trends are to be kept in consideration? These phenomena either make the signal have periodic behavior or they impose a continuous decreasing or augmenting of its value, or maybe both behaviors combined. Common solutions in this case typically try to decompose the signal into its building components to recover seasonality and trend to finally obtain a residual. Such techniques often involve the use of Fourier Transforms and smoothing filters or a method that goes under the name of STL [8]. All these components can then be analyzed in search of anomalies.

In any case, deep learning methods, often based on recurrent architectures[1] are largely employed too when time series are involved. For instance [9] combines LSTMs (a type of recurrent network) with dynamically set thresholds on their output to detect anomalies in telemetry data.

1.2 Action recognition in videos

Relevant to this work are also common techniques to perform action recognition in videos. Albeit many attempts to solve this problem through expert computer vision algorithms coupled with handcrafted features have been made, recent breakthroughs in this field are due to deep learning methods. An example of the former method can be [10], where dense trajectory are extracted from the video, using optical flow and tracking algorithms; later, Histograms of Oriented Gradients (HOG) [11] and other Histograms of Optical Flows (HOF) [12] are applied in regions near tracked trajectories to form a comprehensive descriptor for the action; this is repeated at different scales (i.e. in very simplified terms, zoom levels on the video) to capture different levels of details; lastly, descriptors are grouped in Bag Of Words [13] ensembles during training, to perform the final detection at evaluation time.

Opposite to expert algorithms, are more automatic[2] deep leaning methods. These models showcase a plethora of architectures, ranging from two-stream networks, where there are separate paths for spatial and temporal information up until an information-fusion step before the final output. Therefore alongside the raw input images or even some handcrafted features, such as HOG, these networks are usually fed with Optical Flow information, in an effort to guide how they interpret the time information. This is the case for example of the second version of [14], a two-stream I3D neural network.

Other approaches instead are more data-driven, meaning the networks have to figure out all the necessary descriptive features entirely by themselves, starting from raw images. Temporal information is usually extracted by means of specialized architectures, like three-dimensional convolutions,

[1] Some details are given out in the next chapter.
[2] Since they learn problem-representative features/descriptors by themselves.

for instance as in [14, 15], or recurrent layers.

1.3 Motivations of the proposed system

Figure 1.4: Data Reply IT

This work has been developed as a research project within Data Reply IT, a company belonging to the Reply Holding, focused on big data management and analytics. One of the specializations within Data Reply is on physical security leveraging realtime data analytics. Thus, the company's desire is to see if a lightweight distributed alerting system on CCTV video streams is feasible. The overall system draft Data Reply conceived has many parts, often cloud-based, especially when it comes to training machine learning models. However, the part that has to detect a dangerous situation must run on local machines, which therefore have not unlimited computational power, as a scalable managed cloud solution would have. This is however needed to keep detection as close to realtime as possible, since a centralized solution would run into bandwidth problems when receiving too many video streams at once, not to mention the prohibitive costs that a cloud solution able to digest such huge amount of information would require. Only the signal that something is going wrong, therefore, must be sent to the central component from the localized machine, this way. The central controller can then decide what further actions to take.

1.3.1 Requisites for the system

Because of the imposed conditions, the aim of this work is then to perform a kind of anomaly detection, coupled with action recognition, relevant to the public safety: it must analyze video sources from still CCTV cameras in search for dangerous situations, which go beyond normal behavior contextually to the situation. It must also be able to classify them, or at least their main occurrences, while keeping realtime performances on edge devices (middle range consumer-grade PCs), and a reasonably low false alarm rate.

As it shall be seen, an architecture is proposed to solve the stated problem, however, due to difficulties with the datasets, a conclusive proof-of-concept is not reached.

2
Background Theory

NEURAL NETWORKS are nowadays a common way to extract knowledge automatically[1] from some input data. But albeit powerful and extremely versatile, they are known also to boast high requirements, both in terms of hardware performance, and in the size of data they require, in order to learn meaningful patterns out of them. Combining them with other models may be of benefit, either because the overall system performance improves altogether, while keeping the underlying data the same, or because it may be possible to simplify the cost of the system (in terms of hardware especially, but maybe also in the defining parameters of each model) sacrificing some performance whilst keeping the committed error constrained. This work in particular aims at studying what happens when a neural network has to work alongside a Hidden Markov Model, in the particular context of anomaly detection and classification in time series as shall be seen in later chapters.

This chapter, instead, discusses some needed theoretical background. It will start with a brief introduction on how neural networks are structured, and how they work, shortly focusing then on common techniques to perform time series analysis with them. It will then proceed with introducing the Hidden Markov Model. Finally, it will introduce some result later needed to explain

[1] In the sense that a training process enables the network to figure out the relevant knowledge by itself.

the interplay between these two models, which is the foundation upon this work moves on. Lastly, this interplay happens on a "common ground", which is a probability distribution. The chapter, therefore, ends presenting one suitable distribution, which exhibits some useful properties regarding the probability of error.

2.1 Neural Networks

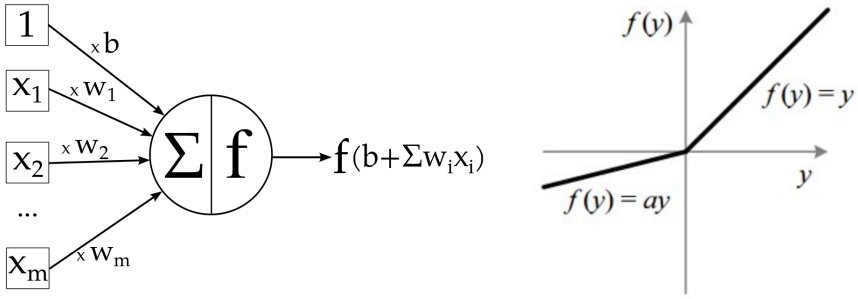


Figure 2.1: Tipical structure of a neuron (right), and a LeakyReLU activation function (left).

The first works on neural networks (for short NN) date back to the early 40s when researchers were attempting at proposing a theoretical model for the human brain. Although far from capturing the complexity of our brain, the resulting neuron model, when repeated and combined a sufficient number of times, turned out to be pretty useful to approximate almost any function.

The basics of a neuron are as follows: it receives several inputs, in Figure 2.1 marked as x_n plus an optional constant of value 1, called bias. Each input is then weighted by the respective parameter w_n or b for the bias, and then they are summed together. It is also worth noting that when a network made of these neurons is being trained, only the weights w_n and b are updated by the learning algorithm. So far the operation is a linear combination of the inputs, hence to introduce nonlinearity to the output, the result of the summation goes through a nonlinear activation function $f : \mathbb{R} \to \mathbb{R}$. The nonlinearity increases the representative capacity of the neural network.

There are many possible activation functions to choose from, each with pros and cons, especially when it comes to training very deep networks, due to a problem known as vanishing gradient. A very common function is the Rectified Linear Activation Unit (ReLU), defined as

$$f(x) = max(0, x) \tag{2.1}$$

because of its simplicity and overall good performance in practical applications, even when it comes to very deep networks. The problem though is that if a neuron receives negative weights, the ReLU outputs a zero. In the gradient backpropagation phase of the training (which is the most common training algorithm for an NN), then, the gradient itself is killed by such zero value, hence the update of the networks's weights may become less efficient. To overcome this problem a simple alternative is the Leaky ReLU function, which is defined as:

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ ax & \text{otherwise.} \end{cases} \tag{2.2}$$

where $a > 0$ is small. Finally, such function avoids problems of vanishing gradients because it doesn't saturate to some limiting values, as is the case for other activations of sigmoidal shape.

Neurons are hardly effective alone. The power of these kinds of models comes from combining neurons together into layers. The more neurons in a layer, the more expressive the resulting neural network becomes. But also the type of connections is relevant in this sense.

The macro-categories current neural networks models fall into are:

- **Deep versus shallow**: theoretically for an NN is sufficient to have an input layer and an output layer. Such a network can already represent a very wide range of functions (actually all boolean functions $g : \{\pm 1\}^m \rightarrow \{\pm 1\}$; refer to [16]), and if it has at least one hidden layer in between, then the expressive power of such models increases to approximating almost every function. But nowadays the vast majority of NN models are of *deep* type, that is, they have $n \gg 1$ hidden layers. This has been proven to vastly increase performances, since each layer crafts a set of features of increasing specificity with respect to the previous, hence downstream

layers have a sort of restricted set of hypothesis to work upon.

- **Fully connected versus convolutional architecture**: NN can have a neuron of some layer $k + 1$ be connected to each of the neurons of the layer k. This is the fully connected case. Such models are very expressive, but also require a lot of training data to tune the huge number of weights thus present, as well as capable hardware to perform the training iterations. Some benefits may come to restrict each layer to share a set of common weights within it, and by having a neuron at layer $k + 1$ be connected only to some selected neurons al layer k. Such connections practically realize a convolution, hence the name of convolutional neural networks (CNN). These models show a speedup in computational performances, while at the same time requiring fewer data to be trained, and are particularly well suited for image/video analysis. The drawback to being aware of is that being the convolution a symmetric operation, so does the CNN consider its input to be as well. Lastly, care has to be taken, because such networks are not invariant even to affine transformation (rotation, scaling, skewing) but translation, not to mention more complex ones (mirroring, perspective deformations, ...).

- **Feedforward versus Recurrent**: the most common NN architectures are all feedforward, that is, neurons at layer k can only connect to neurons at layer $k + 1$. But recently Recurrent Neural Networks (RNN) have gained momentum, since they can learn patterns also in the time dimension, and they are able to exhibit memory as well. In such architectures, a neuron receives as input its own output (at a previous time step), alongside the output of neurons in the previous layers (as is in a normal feedforward NN). The most promising results are obtained by the so-called gated architectures, with particular regards to the Long Short Term Memory (LSTM) one [17]. Gates are specialized neurons with usually sigmoid or hyperbolic tangent activations, that control the amount of memory that can influence the output of the module, or that can control how much of the input of the module can influence its memory states. One of the reasons of the success of such gated modules is that the recurrency makes the learning process unstable, whereas gates allow to stabilize it, other than giving greater control on the memory states themselves.

As mentioned earlier, in order to train an NN, whichever architecture it has, its weights must be updated. By far, the most common training method is to optimize the NN iteratively by gradient descent with respect to a loss function defined on the output. This method takes the name of back-propagation because of the way the gradient is passed along backward through the layers, but the details of it are omitted, being them not relevant for this work. Common losses are the Mean Absolute Error or the Mean Squared Error for regression tasks, whilst multi-class cross-entropy is useful in classification, but there are many more. There are also other techniques to train a NN, such as genetic algorithms, but they are out of the scope of this work as well.

2.1.1 Time analysis with Neural Networks

There are two main methods to exploit temporal information when using neural networks:

1. 3D CONVOLUTIONS. This method generalizes the convolutional networks to act on portions of volumes of the input rather than just portions of planes (or even segments for 1D convolutions). All considerations about convolutions are valid also in this case: 3D CNNs are exceptionally good at extracting features that are invariant to translations in the spatio-temporal dimension, but they're not invariant to scaling or rotations, or in general affine transformations. This is not a limitation though, if training is performed by keeping this in mind. The biggest downside may be on the fact that time memory in these networks is only as big as the time axis of the 3D filter, which is usually small.

2. RECURRENT NETWORKS. These networks exhibit memory properties as already explained, which can also be controlled in the gated version. The advantage is that memory states can carry theoretically infinite time information, although there are several technical limitations that prevent this from being strictly true.

Because of their architecture, 3D convolutional networks are more suited at

identifying patterns that are *locally* common both in time and in space, hence they are well suited for action recognition and classification, for instance, because it is most likely that an action will feature common behaviors with another instance of itself but not with the spatial background or the time context of its execution.

An RNN instead, when subject to new inputs, can leverage potentially all the past history before such input, hence the time context may become relevant as well. It has to be said that the gated RNNs have the ability to forget all irrelevant long term information thanks to specialized "forget" gates, hence if necessary they can behave "similarly" to a 3D CNN. That established, it is still true that RNNs are very well suited at time series regression or forecasting, because of their *infinite* memory.

Each method has pros and cons, and perhaps a mix and match of the two may bring greater performances than just either one alone. Also, the choice of input information most often than not has a great impact as well. Of course though, such considerations are highly application dependent, hence no further general analysis will be carried on them throughout this work. Rather, this work chooses a 3D convolutional approach for mainly two considerations:

1. Its end target is a subfield of action recognition.
2. Hardware limitations. RNN are less efficient, requiring more resources.

2.2 The Hidden Markov Model

> Markov Chains tells us something about the probabilities of sequences of random values, called states, each of which can take values from some set. Hidden Markov Models are based on augmenting Markov Chains.

as pointed out in [2, section A.1]. The augmentation comes from the fact that rather than observing directly the state sequence, only symbols emitted

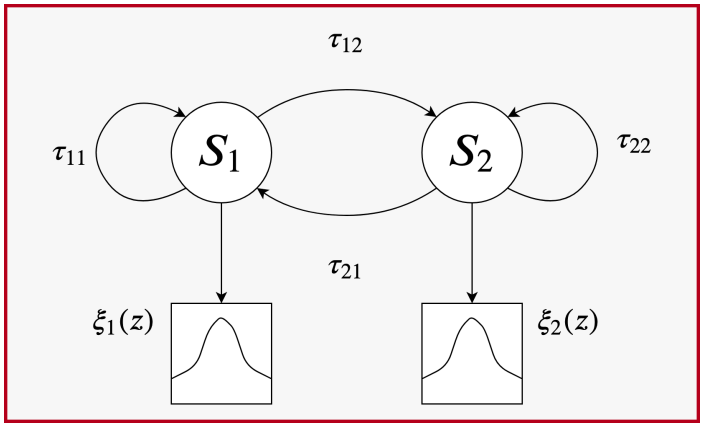


Figure 2.2: A two state Hidden Markov Model

by each state with some probabilities can be observed, whilst the states remain hidden, hence the name of the model. They are defined by the following parameters:

$S = \{s_1, ..., s_N\}$	a set of N states
$\mathbf{T}$	a transition matrix of size $N \times N$, with each entry τ_{ij} representing the transition probability from state i to state j, and such that $\sum_{j=1}^{N} \tau_{ij} = 1$
$Z = \{\mathbf{z}_1, ..., \mathbf{z}_T\}$	a sequence of T observations
$\xi_i(\mathbf{z}_t)$	emission probability of observation $\mathbf{z}_t$ from state s_i
$\pi_1, ..., \pi_N$	initial states probability distribution

Table 2.1: Hidden Markov Model parameters. Adapted from [2, section A.1]

Indeed Hidden Markov Models (HMM) are rather simplistic tools to describe sequences, since they make two strong simplifying assumptions [2]:

1. Markov Assumption: the probability of a particular state only depends on the previous state, and nothing else. Formally

$$P(s_i|s_0, ..., s_{i-1}) = P(s_i|s_{i-1}) \tag{2.3}$$

2. Output Independence: the probability of an observation z_i only depends on the state that produced it, and not any other state or observation. Formally

$$P(\mathbf{z}_i|s_1, ..., s_i, ..., s_T, \mathbf{z}_1, ..., \mathbf{z}_i, ..., \mathbf{z}_T) = P(\mathbf{z}_i|s_i) \tag{2.4}$$

Similarly to [18], with such assumptions, and defining $E = \{\xi_k(\mathbf{z}_t)\}_{k=1}^N$, one can write the probability of an *observation* sequence as

$$P(\mathbf{z}_1, ..., \mathbf{z}_T|s_1, ..., s_T, E) = \prod_{i=1}^{T} \xi_{s_i}(z_i) \tag{2.5}$$

whereas the probability of a *state* sequence is

$$P(s_1, ..., s_T|\mathbf{T}, \pi_1, ..., \pi_N) = \pi_{s_0} \prod_{i=2}^{T} \tau_{s_{i-1},s_i} \tag{2.6}$$

from which follows that the joint probability of observations and state sequences is

$$P(\mathbf{z}_1, ..., \mathbf{z}_T, s_1, ..., s_T|\pi, \mathrm{T}, E) = \pi_{s_0} \prod_{i=2}^{T} \tau_{s_{i-1},s_i} \xi_{s_i}(z_i) \tag{2.7}$$

The overall stochastic process that can be inferred from the observations is obtained by marginalization of (2.7) over all possible state sequences **s**, hence:

$$\begin{aligned} P(\mathbf{z}_1, ..., \mathbf{z}_T|\pi, \mathrm{T}, E) &= \sum_{\mathbf{s}\in\{s_k\}_{k=1}^T} P(\mathbf{z}_1, ..., \mathbf{z}_T, \mathbf{s}|\pi, \mathrm{T}, E) \\ &= \sum_{\mathbf{s}\in\{s_k\}_{k=1}^T} \pi_{s_0} \prod_{i=2}^{T} \tau_{s_{i-1},s_i} \xi_{s_i}(z_i) \end{aligned} \tag{2.8}$$

Now that the model formulation is clear, there are a few common problems that one would want to solve when using an HMM:

- **Learning a model's parameters from observations**. The HMM's parameters to be learned are its transition matrix, the emission probabilities and the starting probabilities for the states. The number of states N instead must be given. Once a sequence of observations is made available,

this objective can be easily attained by means of the Baum-Welch algorithm, an adaptation of the Estimation Maximization paradigm for the HMM. The details of the algorithm are omitted, but it is a type of iterative gradient-based hill-climbing method, hence it suffers from typical problems of such techniques, such as getting stuck in non-optimal local maxima.

- **Finding the likelihood of an observation sequence**. For an HMM with known parameters, it is possible to use the forward phase of the Baum-Welch algorithm to estimate the probability of an observation sequence under the HMM.

- **Decoding the most likely state sequence**. For an HMM with known parameters, it is possible to estimate the best (hidden) state sequence given an observation sequence. The problem to keep at bay, in this case, is an exponential blow-up of possible sequences that must be checked: for an N-state HMM and an input observation sequence of length T, there are N^T possibilities to be evaluated. Luckily, dynamic programming can be used in practice to reduce the computational burden. A very common choice in such sense is the Viterbi algorithm for sequence decoding.

2.2.1 Tying parameters together

As the number of states in an HMM increase, alike described in [19], the number of parameters that must be found during training increases quadratically in the number of states, because of the HMM's transition matrix. This may lead to a lack of data to support such a high number of unknowns. This problem can be solved by splitting the single HMM into multiple ones, and by tying together some of the parameters of the smaller models. Whilst the choice of parameters to tie can be any, usually one would want to link together the emission probabilities of some state into two or more HMMs. This is very useful when different models use a state to represent the same event, while other states in them are dedicated to representing other, more peculiar happenings.

For instance, in speech recognition HMMs have been used for a while (up until RNNs proved themselves more performing). Usually, speech utterances are represented by 3-state HMMs, whose middle state can be the same on many of such models, while the two outer states represent the context in which the same utterance may appear. The common approach is to cluster together any HMM that may share the middle state, by means of specially crafted decision trees, and to give them all the same emission distribution to associate to the middle state.

This work will make use of a similar approach; although, because it is known apriori which state is shared by the small models, it is not required to use decision trees or other forms of clustering to perform state tying. More details will be given in the next chapter.

2.2.2 Probability of Error

Hidden Markov Models are often used for classification tasks, hence the third problem stated above is among the most common to solve with this class of models.

Being a probabilistic model, one can give a formulation for the probability of error P_e. As in [20], assume there are M classes C_i, $i = 1, ..., M$ into which classify observations. A loss can be defined for misclassification, that is to assign class C_j to observation $\mathbf{z}$ when instead its real class is C_i. One possible loss uses the *a posteriori* probabilities $P(C_i|\mathbf{z})$ and is defined for each class as

$$R(C_i|\mathbf{z}) = \sum_{j=1}^{M} e_{ij} P(C_j|\mathbf{z}) \tag{2.9}$$

where e_{ij} is the cost of erroneously classify $\mathbf{z}$ as belonging to C_j instead of its real class C_i. When this quantity is set to

$$e_{i,j} = \begin{cases} 0 & i = j \\ 1 & i \neq j \end{cases} \tag{2.10}$$

(2.9) becomes the probability of error $P_e(C_i|\mathbf{z}) = R(C_i|\mathbf{z})$, and it can be refor-

mulated as

$$\begin{aligned} P_{e,C_i} &= \sum_{j \neq i} P(C_j|\mathbf{z}) = 1 - P(C_i|\mathbf{z}) \\ P_e &= \int P_{e,C(\mathbf{z})} p(\mathbf{z}) d\mathbf{z} \end{aligned} \tag{2.11}$$

Where $C(\mathbf{z})$ is the class assigned to $\mathbf{z}$. The optimal classifier, that achieves P_e in a Maximum a Posteriori (MAP) fashion, implements then the rule

$$C(\mathbf{z}) = C_i \text{ if } P(C_i|\mathbf{z}) = \max_j P(C_j|\mathbf{z}) \tag{2.12}$$

But in order to really minimize P_e, one needs to maximize $P(C_i|\mathbf{z})\ \forall i$, as can be seen in (2.11). Usually, these conditional probabilities depend on the joint distribution of the observations and classes, and can only be estimated from the training set (hence they are fixed for a particular dataset). It will be shown in section 2.3.2 and the next chapter, that there's actually a way to directly maximize $P(C_i|\mathbf{z})\ \forall i$ instead.

2.3 Systems' behavior

In what follows, it is explained the theory developed for this work, and who's behind the method whose implementation is presented in greater detail in a later chapter. For the moment, let's assume the proposed systems is composed of two probabilistic models λ and θ, each with (learnable) parameters that define them uniquely. How do they interact together when they share the same statistical manifold $\mathbb{Z}$ for their target distribution?

2.3.1 The minimax game

It turns out that one of the possibilities they have is to play a *minimax game* with objective function

$$V(\theta, \lambda) = \mathbb{E}_{\mathbf{z} \sim P_\theta} \left[log \left(\frac{1}{p_\lambda(\mathbf{z})} \right) \right] \tag{2.13}$$

where $P_\lambda(\mathbf{z}) = \int_{\mathbf{z}} p_\lambda(\mathbf{z})d\mathbf{z}$ and $P_\theta(\mathbf{z}) = \int_{\mathbf{z}} p_\theta(\mathbf{z})d\mathbf{z}$ are the respective target distributions.

An immediate result is readily available:

Theorem 1 *In the unique equilibrium point, assuming independence of samples $\mathbf{z} \in \mathbb{Z}$, it is verified that $P_\lambda = P_\theta$ and $\mathbb{E}_{\mathbf{z}\sim P_\theta}[log(p_\theta(\mathbf{z}))]$ is maximum with respect to θ.*

Proof.
The minimax game's objective function can be written as

$$\begin{aligned}\min_\theta \max_\lambda V(\theta,\lambda) &= \min_\theta \max_\lambda \mathbb{E}_{\mathbf{z}\sim P_\theta}\left[log\left(\frac{1}{p_\lambda(\mathbf{z})}\right)\right] = \\ &= \min_\theta \max_\lambda \int_{\mathbf{z}} p_\theta(\mathbf{z}) log\left(\frac{1}{p_\lambda(\mathbf{z})}\right) d\mathbf{z} \\ &\quad + \int_{\mathbf{z}} p_\theta(\mathbf{z}) log(p_\theta(\mathbf{z}))d\mathbf{z} - \int_{\mathbf{z}} p_\theta(\mathbf{z}) log(p_\theta(\mathbf{z}))d\mathbf{z} = \\ &= \min_\theta \max_\lambda -D_{KL}(P_\theta||P_\lambda) - \mathbb{E}_{\mathbf{z}\sim P_\theta}[log(p_\theta(\mathbf{z}))]\end{aligned} \tag{2.14}$$

At λ's turn, its only way to maximize $V(\cdot)$ is to minimize $D_{KL}(\cdot)$, being this quantity not only always nonnegative, but also the only one λ has direct access to. This is possible only for $P_\lambda = P_\theta$, where $D_{KL}(\cdot) = 0$.

When is θ's turn instead:

$$\min_\theta -D_{KL}(P_\theta||P_\theta) - \mathbb{E}_{\mathbf{z}\sim P_\theta}[log(p_\theta(\mathbf{z}))] = \max_\theta \mathbb{E}_{\mathbf{z}\sim P_\theta}[log(p_\theta(\mathbf{z}))] \tag{2.15}$$

Hence there is an equilibrium point, with the required properties. This point is also unique since $V(\cdot)$ only has this saddle point with respect to λ and θ: as mentioned, λ can only minimize D_{KL}, which has a single global minima, whereas the expected value in (2.15) is always unique for each possible θ, moreover no two different θ' and $\hat{\theta}$ can have the same value for (2.15), because

trivially

$$\begin{aligned} &\mathbb{E}_{z\sim P_{\theta'}}[g(z)] = \mathbb{E}_{z\sim P_{\hat{\theta}}}[g(z)] \\ &\iff \int_z g(z)p_{\theta'}(z)dz = \int_z g(z)p_{\hat{\theta}}(z)dz \\ &\iff P_{\theta'} = P_{\hat{\theta}} \\ &\iff \theta' = \hat{\theta} \end{aligned} \tag{2.16}$$

□

This theorem alone is not all that useful, because the equilibrium distribution cannot be controlled. But interpreting Theorem 1 as the stage of an infinite horizon repeated game may bring some benefits towards such end. This is exactly the result the following theorem explains:

Theorem 2 *Let $k = 1, 2, ...$ be the number of iterations for which the stage game is repeated. Let also $g_k : \mathbb{R}^N \to \mathbb{R}^N$ be a sequence of measurable functions (so that $\mathbf{y} = g_k(\mathbf{z})$ is a r.v. itself) such that $lim_{k\to\infty} g_k(\mathbf{z}) = [IdentityFunction\ \mathbf{y} = \mathbf{z}]$. Then for $k \to \infty$ the infinite horizon repeated game has an equilibrium point where $P_\lambda(\mathbf{y}) = P_{g_\infty}(\mathbf{y}) = P_\theta(g_\infty^{-1}(\mathbf{y})) = P_\theta(\mathbf{z})$ and $\mathbb{E}_{\mathbf{z}\sim P_\theta}[log(p_\theta(\mathbf{z}))]$ is maximum with respect to θ.*

Proof.
The objective function of the repeated game is only slightly modified due to $g_k(\cdot)$ with respect to Theorem 1:

$$\min_\theta \max_\lambda \quad \mathbb{E}_{\mathbf{y}\sim P_{g_k}}\left[log\left(\frac{1}{p_\lambda(\mathbf{y})}\right)\right] \tag{2.17}$$

Due to Theorem 1, though, at each iteration k, for the stage game, is still true that it converges toward the unique equilibrium point given by

$$\begin{aligned} P_\lambda(\mathbf{y}) &= P_{g_k}(\mathbf{y}) \\ \max_\theta \mathbb{E}_{\mathbf{y}\sim P_{g_k}}\left[log(p_{g_k}(\mathbf{y}))\right] &= \max_\theta \mathbb{E}_{\mathbf{y}\sim P_\theta(g_k^{-1}(\mathbf{y}))}\left[log\left(\frac{dP_\theta(g_k^{-1}(\mathbf{y}))}{d\mathbf{y}}\right)\right] \end{aligned} \tag{2.18}$$

where the measurability of g_k has been leveraged. Because the equilibrium point is unique at each stage, for Proposition 2 in Appendix A, also the re-

peated game has the same equilibrium point. Finally, the equality

$$P_\lambda(\mathbf{y}) = P_{g_\infty}(\mathbf{y}) = P_\theta(g_\infty^{-1}(\mathbf{y})) = P_\theta(\mathbf{z}) \tag{2.19}$$

is a direct consequence of the measurability of g_k and the convergence of the sequence towards the identity function, as $k \to \infty$.

□

The function $g_k(\cdot)$ practically enables gradually shaping the equilibrium distribution in a desirable way. For this reason, while formally the same, the equilibrium point of Theorem 2 is rather different from that of Theorem 1, on a practical, operational, level.

The following lemma instead gives an operative way to find the equilibrium point for θ:

Lemma 1 $\max_\theta \mathbb{E}_{x\sim P_\theta}[log(p_\theta(x))] \iff \max_\theta \sum_n log(p_\theta(x_n))$ *for* $n \to \infty$.

Proof.
From [21] the Monte Carlo Estimator for $\mathbb{E}_{x\sim P_\theta}[log(p_\theta(x))]$ is

$$\frac{1}{N}\sum_{n=1}^{N}\left[\frac{log(p_\theta(x_n))p_\theta(x_n)}{p(x_n)}\right] \tag{2.20}$$

Indeed it is

$$\begin{aligned}
&\mathbb{E}_p\left[\frac{1}{N}\sum_{n=1}^{N}\left[\frac{log(p_\theta(x_n))p_\theta(x_n)}{p(x_n)}\right]\right] = \\
&= \frac{1}{N}\sum_{n=1}^{N}\mathbb{E}_p\left[\frac{log(p_\theta(x))p_\theta(x)}{p(x)}\right] = \\
&= \frac{1}{N}\sum_{n=1}^{N}\int\left[\frac{log(p_\theta(x))p_\theta(x)}{p(x)}\right]p(x)dx = \\
&= \int log(p_\theta(x))p_\theta(x)dx = \\
&= \mathbb{E}_{p_\theta}\left[log(p_\theta(x))\right]
\end{aligned} \tag{2.21}$$

where $P(x) = \int_{-\infty}^{x} p(x)dx$ is an arbitrary distribution, hence it can be chosen equal to $P_\theta(x)$, obtaining the estimator

$$\frac{1}{N}\sum_{n=1}^{N}[log(p_\theta(x_n))] \tag{2.22}$$

Such estimator equals the estimated quantity only for $n \to \infty$. In this situation

$$\max_\theta \mathbb{E}_{x\sim P_\theta}[log(p_\theta(x))] = \max_\theta \frac{1}{N}\sum_{n=1}^{N}[log(p_\theta(x_n))] = \max_\theta \sum_n log(p_\theta(x_n)) \tag{2.23}$$

from which the Lemma follows.

□

The Lemma allows to operatively estimate the value for the equilibrium point of θ, in particular as a Maximum Log-Likelihood estimation. There are several efficient algorithms that perform such estimation, a notable one being the Expectation-Maximization (EM) family of algorithms.

It still remains to prove that the equilibrium point is actually reached by the game, since it may be the case that this is not true in some games and or situations. The proof is adapted from [22, proposition 2]:

Proposition 1 *Given a sufficient number of samples from $\mathbb{Z}$, the infinite horizon repeated minimax game converges towards its unique equilibrium point.*

Proof.
Since $D_{KL}(P_{g_k}||P_\lambda)$ is convex with respect to both distributions, with an argument similar to [22, proposition 2], it follows that actually P_λ converges toward P_{g_∞} (which is held fixed, similarly to D in the reference) for $k \to \infty$. Lastly, thanks to Lemma 1, even θ converges as expected at each iteration k, given $n \to \infty$ samples coming from $\mathbb{Z}$.

□

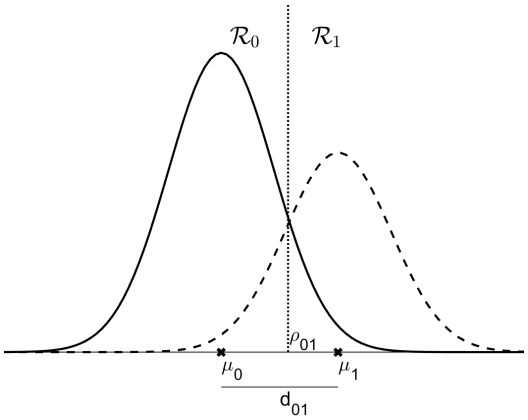


Figure 2.3: A one-dimensional binary Gaussian constellation.

2.3.2 Gaussian constellation

The concept of a constellation, especially associated with Gaussian probabilities, is mainly employed within the communication engineering world, where it is used for digital signal modulation, however, its properties about the error probability can be of interest in this context.

Figure 2.3 depicts a binary constellation, with non uniform probabilities for its components. In what follows all Gaussians probabilities associated to each component are assumed of equal variance, and in the multivariate case, it is also assumed independence among dimensions, so that each Gaussian can be characterized as $\mathcal{N}(\mu_i, \sigma^2\mathbf{I})$. The MAP criterion assigns a random point $\mathbf{z}$ to the Gaussian that maximizes its a posteriori probability, which can be expressed as $P(C_i|\mathbf{z}) = P(\mathbf{z}|C_i)P(C_i)/P(\mathbf{z})$. Here $P(\cdot|C_i)$ is Gaussian, whereas $P(C_i)$ are the a priori class probabilities. $P(\mathbf{z})$ is instead irrelevant for the maximization criterion. This translates in finding optimal regions that partition the space such that $\mathcal{R}_i \cap \mathcal{R}_j = \emptyset \; \forall i, j$ and $\mathbb{Z} = \cup_i \mathcal{R}_i$. When a point falls within region $\mathcal{R}_i$, then class C_i is assigned to it. In Figure 2.3, optimal regions $\mathcal{R}_0$ and $\mathcal{R}_1$ are delimited by ρ_{01}. In such point it must be true that

$$P(\rho_{01}|C_0)P(C_0) = P(\rho_{01}|C_1)P(C_1) \tag{2.24}$$

With simple enough computations, it turns out that

$$\rho_{01} = \frac{d_{01}}{2} + \frac{\sigma^2}{d_{01}} ln\left(\frac{P(C_0)}{P(C_1)}\right) = \frac{d_{01}}{2} + \epsilon_{01} \tag{2.25}$$

where $d_{01} = \sqrt{(\mu_0 - \mu_1)^2}$ is just the distance between two components of the constellation. Notice that as $d_{01} \to \infty$ the additional information coming from knowing the a-priory class probabilities becomes negligible in determining the regions' edge. In the binary case, then, the error probability is

$$\begin{aligned} P_e &= 1 - P_c = 1 - [P(z \in \mathcal{R}_0|C_0)P(C_0) + P(z \in \mathcal{R}_1|C_1)P(C_1)] = \\ &= 1 - \left[P(z \leq \frac{d_{01}}{2} + \epsilon_{01}|C_0)P(C_0) + P(z \leq \frac{d_{01}}{2} - \epsilon_{01}|C_1)P(C_1)\right] = \\ &= Q\left(\frac{\frac{d_{01}}{2} + \epsilon_{01}}{\sigma}\right)P(C_0) + Q\left(\frac{\frac{d_{01}}{2} - \epsilon_{01}}{\sigma}\right)P(C_1) \end{aligned} \tag{2.26}$$

Where symmetry considerations have been exploited as well. $Q(\cdot)$ is the complementary error function for a Gaussian probability instead.

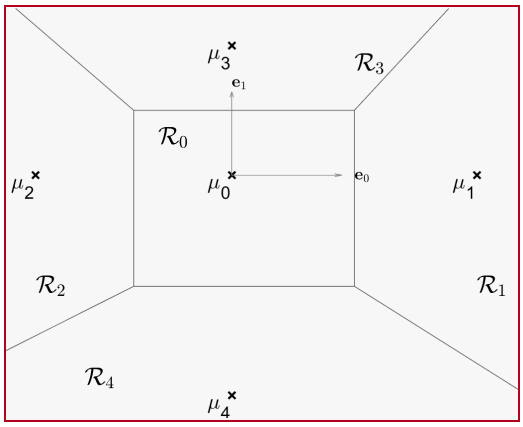


Figure 2.4: The two-dimensional version of the constellation employed in this work.

To tackle the case of $M+1$ classes useful for this work, the constellation that is actually considered is constructed as follows:

- There is a central component, representing class C_0, which also has the highest probability of all classes ($P(C_0) > P(C_i)\ \forall i = 1, ..., M$), placed at the origin of $\mathbb{R}^{M/2}$.

- all the other M classes (M ideally even) are placed in pairs along the eigenvectors of $\mathbb{R}^{M/2}$, each member of a pair sitting opposite to the other with respect to C_0, along a chosen eigenvector.

Figure 2.4 has a depiction of the constellation for the case $M+1=5$. This particular constellation has the advantage to put the central component, since it is the most probable one, against all the others.

Finding P_e for a non-binary constellation, especially in closed form, is much more difficult, hence, as explained in [23], it is convenient to resort to an upper bound. An easy method leverages the mutually exclusive events of binary errors between two classes, and the union bound. This often results in a bound not very tight, however, due to the low probability assumed for each non-central component, in this case, most of its inefficiency comes from considering multiple times the central region. But being the most probable one, the central region has also a low probability of an error, hence for such constellation the overall lower bound, albeit loose, is acceptable. This is even more true, because, in the context of this work, it can be made arbitrarily small, pushing the real P_e close to zero, by optimizing $d_{0k}\ \forall k = 1, ..., M$.

With such considerations, the upper bound for the error probability of the $M+1$ classes instance is

$$P_e \leq \sum_{k=1}^{M} \left[Q\left(\frac{\frac{d_{0k}}{2} + \epsilon_{0k}}{\sigma} \right) P(C_0) + Q\left(\frac{\frac{d_{0k}}{2} - \epsilon_{0k}}{\sigma} \right) P(C_k) \right] \tag{2.27}$$

At this point, theoretically letting $d_{0k} \to \infty\ \forall k$ would make sure that $P_e \to 0$. If one just wants to be sure that P_e for the constellations stays below a threshold h, an even looser upper bound can be used:

$$P_e \leq MQ\left(\frac{\frac{d_{0\overline{k}}}{2} - \epsilon_{0\overline{k}}}{\sigma} \right) \tag{2.28}$$

where $\overline{k}$ represents the index of the second most probable class. Hence

$$P_e \leq h \implies \begin{cases} d_{0\overline{k}} \geq Q^{-1}(hM) + \sqrt{[Q^{-1}(hM)]^2 + 2\sigma^2 ln\left(\frac{P_0}{P_{\overline{k}}}\right)} \\ d_{0\overline{k}} \leq d_{0k} \quad \forall k = 1, ..., M \end{cases} \tag{2.29}$$

Where $Q^{-1}(\cdot)$ is the inverse of the complementary error function. Of course, being (2.28) very loose, the resulting $d_{0\bar{k}}$ will be much higher than what strictly necessary, but this is uninfluential for this work.

3
System Architecture

An implementation of what has been found previously is presented in this chapter. With the models that have been chosen, though, a perfect implementation is not possible: some suboptimalities are present and must be dealt with.

This chapter therefore first introduces how the flow of input data is organized. It then proposes an algorithm to implement what is explained in the Theory Chapter, making some considerations on what influence neural networks architecture have on the system in the meantime. Lastly, it makes some clarifications on how each piece works together with the others.

3.1 Input Data Flow

This works deals with time series, hence the need to organize the input data flow in a structured manner. Theory wise, no assumption is made on the dimensionality of the data, other than it must have a dimension interpretable as time; though when it boils down to the implementation, data are assumed to come in video format (hence it is 3-dimensional: time $\times$ image width $\times$ image height). From now on, input data will be referred to as "video" without loss of generalization.

The video stream is divided into contiguous non-overlapping pieces, currently exactly one second long, and those pieces shall be referred to as *segments*. Each segment is then considered to be made up of t *units*, where for the moment $t = 5$.

The objective of the work is to classify each unit into a class, either representing normality or one of several abnormalities. In the following, the normal class may be referred to also as negative, whereas the others as positive. A rather strong but realistic assumption must be made at this point:

Assumption 1 *each unit is strictly classifiable into one and only one class; moreover, an abnormal unit can only be followed either by another one coming from the* same abnormal class *or by a normal unit.*

Also, if needed, every segment that contains at least one abnormal unit is considered to be the same itself, otherwise it is normal. Assumption 1 restricts the range of "legal" segments, visible in Table 3.1. Whilst this can not be leveraged during training to optimize it, it can be used to filter out false detection during inference. Further work may investigate also if there is something to gain in considering the order in which segments can follow each other since Assumption 1 also imposes restrictions on it.

NORMAL	ABNORMAL
−−−−−	−−−−+, −−−++, −−+++, −++++
	+++++
	++++−, +++−−, ++−−−, +−−−−

Table 3.1: Visual representation of each allowed segments. Normal units are marked with "−", abnormal with "+" instead.

To conclude, it should be pointed out for clarity that one unit does not correspond to one video frame, but to a group of them. How large is the group depends on the frames per second (FPS) property of the video. Currently, the videos used all have 30 FPS each, hence a unit should map to a group of 6 frames. In reality, it maps only to 3 frames but sampled in an interleaved fashion so as to cover the same time span of 6 of them (this is because of the need for data augmentation, explained in the next Chapter).

3.2 Minimax Game implementation

The following algorithm provides an implementation of Theorem 2:

Algorithm 1 Infinitely repeated minimax game

Input
- $\lambda \leftarrow$ neural network
- $\theta \leftarrow$ hidden markov model
- $x \leftarrow$ input videos
- $s' \leftarrow$ ground truth labels

1: procedure training(λ, θ, x, s')
2: for $k = 1, 2, ...$ do
3: $z \leftarrow \lambda(x), z \in \mathbb{Z}$ ▷ z is the NN prediction about the input x
4: Train HMM on z, $z \in \mathbb{Z}$
5: $s \leftarrow \theta(z)$ ▷ s are the predicted labels (states of the HMM)
6: if $s = s'$ then return ▷ the g_k function of Theorem 2 has become an identity
7: Train λ on s' and x to output a Gaussian constellation on $\mathbb{Z}$

Algorithm 1 takes as input a neural network and a Hidden Markov model, with roles of λ and θ respectively. Also, the videos and their unit-level labels are provided as input alongside the models. The repeated game can then start, but it must be noticed that the minimax stage game is laid out in a different order than what is hinted at in Theorem 1 and 2: the actual game starts with training λ and ends when θ is asked to predict the videos' labels. This is just an implementation convenience, as it can be assumed that in the very first iteration the NN has already been trained, finding itself with random weights as a result.

The g_k function Theorem 2 talks about, instead, corresponds to the known and invertible mapping between s and s', for each turn k. The convergence of such mapping towards the identity is a result of the training procedure of λ. Now, Assumption 1 makes sure there is a strict one to one mapping from s to $z \in \mathbb{Z}$ and s' to $z' \in \mathbb{Z}$ (the ground truth sample coming from the target distribution) as well. Therefore as s converges towards s', so does z with z'. When this happens, both λ and θ share also the same distribution on the latent space

$\mathbb{Z}$. Unfortunately, this is not strictly true because of some suboptimalities that come from the choice of an HMM as θ: it goes further than interpreting the underlying distribution as a static Gaussian constellation since it also considers the transition probabilities among components. Luckily, Assumption 1 enforces also low transition probabilities between the normal class C_0 and all the other abnormal classes C_i, $i = 1, ..., M$, while there aren't at all transitions between two abnormal classes. This means that the approximation error of modeling the static Gaussian constellation outputted by λ by means of θ is kept somewhat small. In turn suboptimality is traded with a greater easiness of implementation by utilizing this architecture.

3.3 Neural Network Architectures

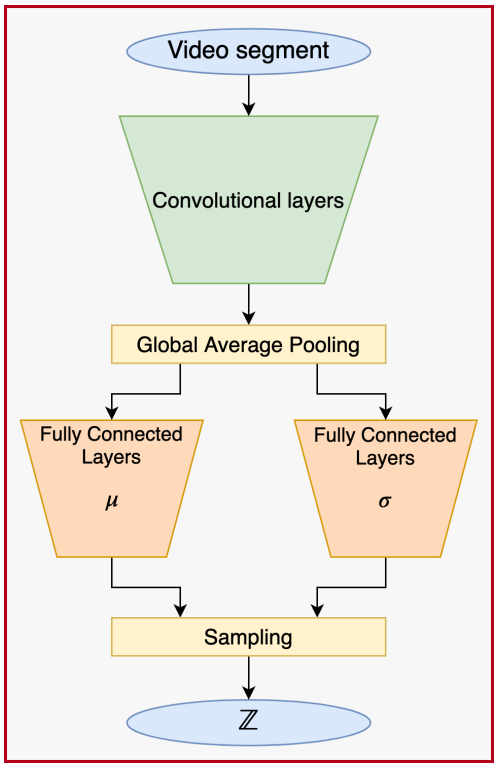


Figure 3.1: Neural Network's architecture.

The objective the NN architecture must accomplish is to take video segments as input and produce as many generative parameters for Gaussians (μ_i and σ_i for $i = 0, ..., M$) as there are units within a segment. Many architectures were evaluated, ranging from simple feedforward networks to recurrent ones. At last, the architecture stabilized onto the following structure:

- A 3D convolutional part that performs feature extraction from video segments.
- Two distinct fully connected parts that take the feature vector found before and output respectively all the needed μ and σ.

A sketch is visible in Figure 3.1. The convolutional component mainly takes inspiration from [24], because it allows for great modularity, extending it also to use 3D convolutions. Of great importance is also the concept of residual blocks, that allow keeping the vanishing gradient problem, particularly evident for very deep networks, at bay. Other architectural considerations are adopted as well. In particular [25] introduces the use of convolutions with kernel 1 to allow for increased computational speed, useful for realtime performances, while they are also handy to keep the number of trainable parameters under control. At the same time, reduction/increase of the number of convolutional filters creates bottlenecks in the data flow within the network, forcing it to learn meaningful representation efficiently, softening learning problems due to a reduced number of weights. In [21, 26] instead is explained how the use of uncorrelated Gaussians samples as the output of the network helps in regularizing the results, while at the same time favoring disentanglement of features. All these considerations are embedded in the network architecture. There are many other improvements that can be made, for instance providing multiple propagation paths within the network, to encourage for filter uncorrelatedness, but they are left for future development. As fas as why 3D convolutions have been chosen, as mentioned in the previous Chapter, it is because of the locality of the features they extract, as well as computational resources availability and constraints.

3.3.1 Architecture Details

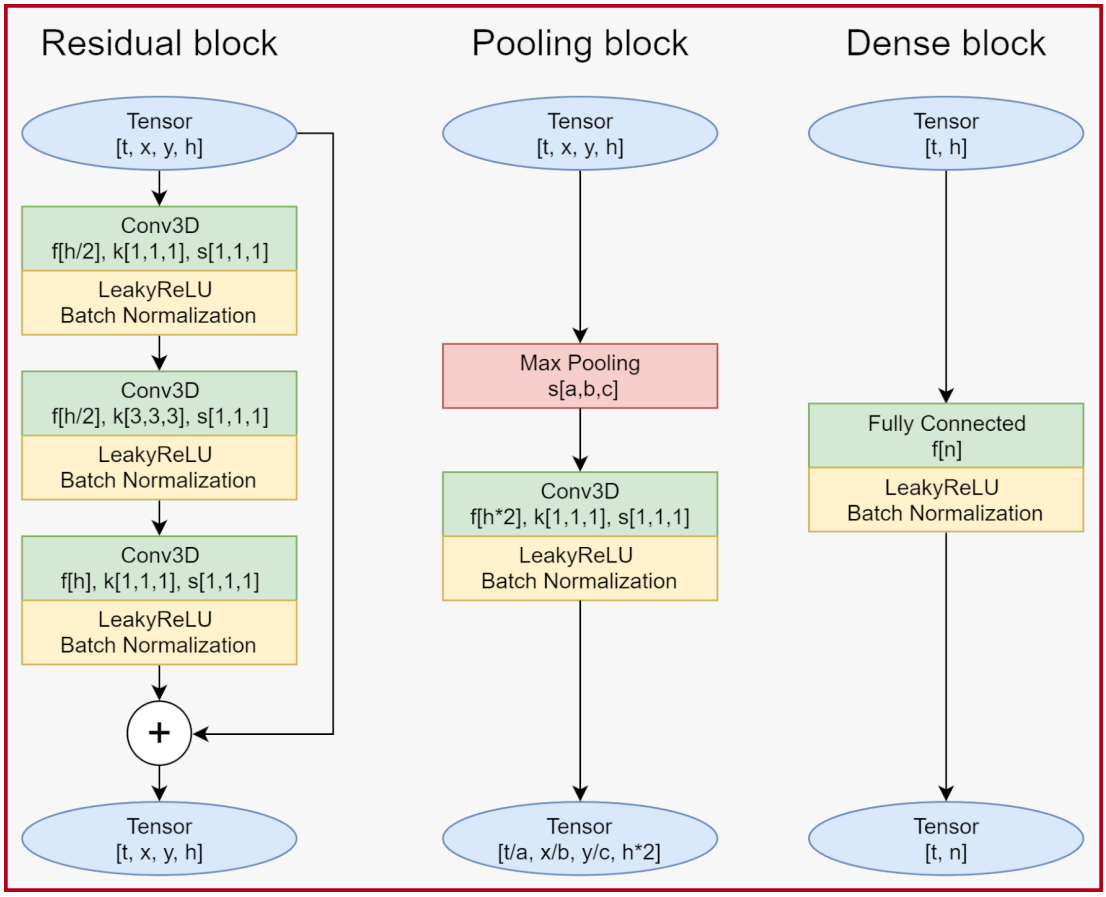


Figure 3.2: Neural Network's building modules. Naming conventions are the same as explained for Table 3.2.

The whole architecture only uses 3 types of modules:

1. The main building block of the convolutional architecture, visible in Figure 3.2 (left), is the residual block, from [24]. These blocks learn the input/output mapping $y = H(x) - x$ rather than just $y = H'(x)$, hence $H(x) = H'(x) + x$. The weights in the block have to be adapted only to learn the residual with respect to the identity $y = x$, and this turns out to be most often an easier problem to solve on its own: rather than to learn a completely new function $H(x)$ from scratch, the optimizer that performs gradient descent has a starting point to begin with, x

itself. Moreover, the gradient backpropagated through the identity connection doesn't get multiplied by possibly small weights, keeping its intensity unchanged for later layers to benefit from, adding quality to the learning process, and allowing for deeper architectures. Therefore the block presents two paths: one is cut-through and realizes the identity mapping $y = x$, the other one has trainable weights on it, and learns the residual function. At last the two paths are added together. The weighted path also presents convolutions with kernel 1 before and after the main convolutional layer, for computational efficiency considerations, but whose role is also to perform bottlenecking of the information by changing the number of filters.

2. Another module, visible inf Figure 3.2 (center), performs instead spatial/temporal dimensionality reduction by means of a max-pooling layer. This operation divides the input volume into a grid, and for each cell of the grid, it takes the maximum value within and copies it to the output, thus reducing the dimensions of the latter. The pooling layer is followed by convolution with kernel 1 to allow the network to fine-tune the result, without impacting performances.

3. The last module employed, as pictured in Figure 3.2 (right) is just a normal fully connected layer, where each neuron has a connection with all the neurons of the previous layer.

Each layer within a module is also followed by a LeakyReLU activation function and by a batch normalization layer, whose role is to standardize the output of a layer during training only. This is a measure to stabilize the learning process [27]: at each iteration of gradient descent, every layer changes its weights; this means that at the next iteration every layer not only has to adapt the weights to converge towards the minimum of the objective function but also to account for all other layers modifications; this slows down the training, and batch normalization aims at reducing this effect by means of standardizing the output of each layer. The only exception is the very last fully connected layer of each of the two separate paths, where the activation is linear and there isn't any batch normalization, being these layers the output ones.

Lastly, to avoid boundary problems, convolutions are padded using symmetric extensions of video volumes boundaries, rather than using the default

BINARY	
Conv3D (f[8], k[5,5,5], s[1,2,2])	
Pooling Block (f[8], s[1,2,2])	
1 Residual Block (f[16])	
Pooling Block (f[16], s[3,2,2])	
1 Residual Block (f[32])	
Global Average Pooling	
Fully Connected (f[32])	Fully Connected (f[32])
Fully Connected (f[16])	Fully Connected (f[16])
Fully Connected (f[1])	Fully Connected (f[1])
Gaussian Sampling	
MULTICLASS	
Conv3D (f[16], k[5,5,5], s[1,2,2])	
Pooling Block (f[16], s[1,2,2])	
1 Residual Block (f[32])	
Pooling Block (f[32], s[3,2,2])	
1 Residual Block (f[64])	
Global Average Pooling	
Fully Connected (f[64])	Fully Connected (f[64])
Fully Connected (f[32])	Fully Connected (f[32])
Fully Connected (f[1])	Fully Connected (f[1])
Gaussian Sampling	

Table 3.2: Neural networks architectures parameters for the binary and multiclass classification problems of the generated dataset.

zero-padding. This is especially useful in the time dimension, to enforce a more consistent behavior of the convolution near borders of two contiguous video segments.

The following table sums up the architecture for two neural networks used on a generated dataset (see next Chapter), for the binary classification and the multiclass cases respectively. The notation f[$\star$] highlights how many filters (or units in a fully connected module) there are at the output of the block. For the convolutional part of the networks, k[$\star$] denotes the dimensions of the kernel where present, with time being the first dimension. Instead, s[$\star$] denotes the strides, with the same convention as for the kernel.

Both networks receive as input video segments of size $15 \times 28 \times 28$ where

the first number is the time dimension, and output 5 points belonging to $\mathbb{Z}$, the Gaussian constellation space, grouped in a matrix [1] of size 5×1. Here the first number also denotes time, as interpreted by the HMM that follows.

3.3.2 Variations on architectures

As mentioned, many variations on the architecture were evaluated. The most notable ones are:

- **Different breaking points**: different positions for the point where the architecture splits and creates two paths, after the Global Average Pooling layer, have been analyzed. No particular benefits of moving it lower were observed over the proposed architecture, though. Quite the contrary happened by moving it higher, where degradation of performances was observed. This is expected since the two resulting sub-networks have too little common points, and end up threating the same input video segment in an uncorrelated manner, whereas this is not true: each path must find mean and variance, respectively, for the same element, hence they are correlated in some way.

- **Different activation functions**: at some point Leaky ReLU were swapped with classical hyperbolic tangent ("tanh") activations, mostly to try to counter a problem of exploding activations the NN model suffers a bit from. However, they greatly reduced performances since they often saturated in the small-gradient regions, preventing the NN from learning adequately.

- **Number of learnable parameters**: being the NN structure modular, especially in the convolutional part, it is somewhat easy to change the number of learnable parameters, from some hundreds of thousands all the way up to tens of millions. The models in Table 3.2 have approximately 25k and 57k parameters respectively.

To try to solve some problems on the real dataset, aside from tinkering with the above architectural changes, two other methods where considered:

[1] More correctly a tensor, using TensorFlow's naming convention.

- **Auto Encoder**: after the Global Average Pooling layer, a third path, other than the two Fully Connected ones, was added. This path had the same functions as the decoder portion of an Auto Encoder.
- **Transfer learning**: transfer learning from pre-trained I3D [14] and C3D [15] nets have been attempted. These networks replaced the convolutional part plus some of the first Fully Connected layers of the architecture.

Since this deserves an in-depth explanation, more will be said in the next Chapter.

3.3.3 Neural Network's Loss

Algorithm 2 KL divergence loss

Input
$\mu_1, \sigma_1 \leftarrow$ ground truth values
$\mu_2, log\_\sigma_2 \leftarrow$ estimated values from the NN

procedure KL_loss($\mu_1, \sigma_1, \mu_2, log\_\sigma_2$)
  $\sigma_2 \leftarrow \exp(log\_\sigma_2)$
  $d \leftarrow \text{Round}\left(\frac{M}{2}\right)$ ▷ M + 1 is the dimension of the constellation
  $log\_det_1 \leftarrow \sum_{k=1}^{d} \log(\sigma_{1,k})$
  $log\_det_2 \leftarrow \sum_{k=1}^{d} log\_\sigma_{2,k}$
  $log\_det \leftarrow log\_det_2 - log\_det_1$
  $trace \leftarrow \sum_{k=1}^{d} \sigma_{1,k}/\sigma_{2,k}$
  $prod \leftarrow \sum_{k=1}^{d} (\mu_{2,k} - \mu_{1,k})^2/\sigma_{2,k}$
  $KL \leftarrow \frac{1}{2}(trace + prod - d + log\_det)$
  return KL

The NN must output, for each unit of a video segment, matching μ and σ with the ones provided by the Gaussian Constellation. This can be accomplished by defining a loss over how much the output of the NN differs from the target value. A mean squared error loss is often employed for regression problems such as this one, but in this particular instance, it is also possible to use directly the Kullbach-Leibler divergence between multivariate Gaussians

since it has a closed-form expression. This not only more closely matches the result of Theorem 2, but also provides the optimizer for the NN with direct knowledge of the difference between two Gaussians, rather than just an indication of how good the NN is performing.

The KL divergence for the multivariate case is:

$$D_{\mathrm{KL}}(\mathcal{N}_1 \parallel \mathcal{N}_2) = \frac{1}{2}\left(\mathrm{tr}\left(\Sigma_2^{-1}\Sigma_1\right) + (\mu_2 - \mu_1)^{\mathsf{T}}\Sigma_2^{-1}(\mu_2 - \mu_1) - d + \ln\left(\frac{|\Sigma_2|}{|\Sigma_1|}\right)\right) \quad (3.1)$$

Finally, because it is assumed that $\Sigma_i = \sigma_i I$, i.e a diagonal matrix, the loss is implemented as in Algorithm 2. The σ_2 value is interpreted as $\log(\sigma_2)$ by the NN to force it's non-negativity without imposing constraints on the activation of the last fully connected layer.

3.4 Hidden Markov Model Architecture

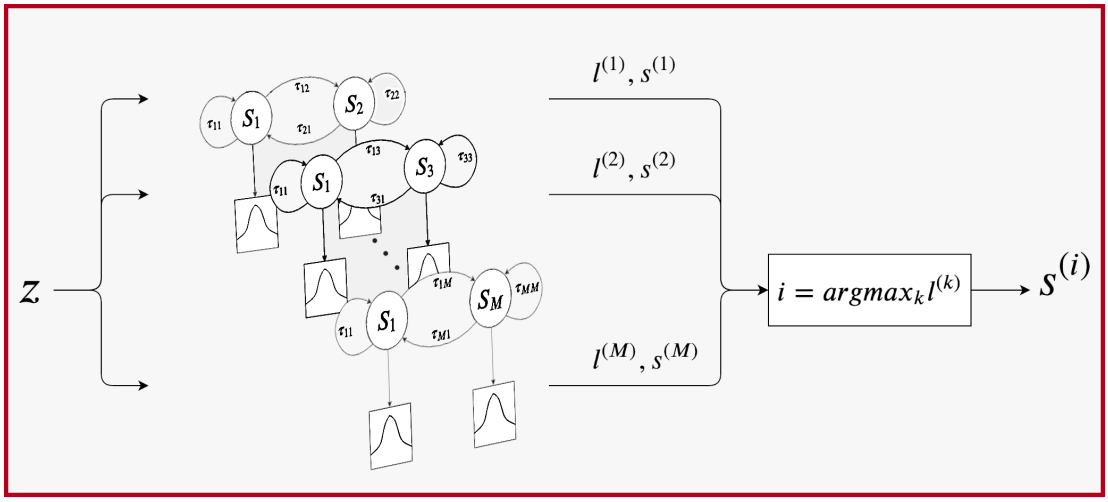


Figure 3.3: Inference procedure for M tied state HMMs: z is the input sequence, whereas $s^{(i)}$ is the predicted state sequence, selected from the highest scored HMM i, in terms of the log-probability $l^{(i)}$.

The Hidden Markov Model architecture is fairly simple, thanks to Assumption 1. For a two classes classification problem it is sufficient to use a two-state HMM as shown in Figure 2.2. Indeed, the first state represents the negative class, whereas the other one the positive class. Transition probabilities

towards the states themselves, that is τ_{11} and τ_{22}, are expected to be very high, close to 1; the others are of course low, somewhat close to 0.

To extend the HMM to the multiclass instance, one would need to add one state per extra positive class. This would result in a quadratic explosion of trainable parameters, because of the transition matrix. An explosion that is entirely wasteful, since the matrix is very sparse, having only the main diagonal and the first row and column different from zero (because of Assumption 1).

To exploit the knowledge of this sparsity, it is sufficient to notice that the normal class fares against only one of the positive classes at a time. Hence it is possible to create M 2-state HMM, where M is the number of abnormal classes: each one of these little models has one state to represent the normal class, an the other to represent its positive class of reference. It is then sufficient to make sure the representation of the negative class is consistent among each model. This translates simply in making sure the emission probability associated with the elected state is the same in each model. This is a very simple implementation of the state tying strategy briefly explained in Subsection 2.2.1, especially because which state must be tied is known, eliding the need for a decision tree (that must be trained as well). Just a little care must be taken in training separately each HMM on inputs coming only from the normal class plus the respective positive class.

During inference, instead, each HMM has to be run on the same input sequence: the one yielding the highest fit is selected to explain the data. A representation is visible in Figure 3.3.

A note on the training is due at this point: as mentioned in Chapter 2, HMMs are trained by means of MAP criterion. Because their input data comes from the latent space $\mathbb{Z}$, whose distribution gradually resembles a Gaussian constellation, it is possible to increment their classification performance. The constellation is built to maximize $P(z|C_i)\ \forall i$, which has a Gaussian shape. This ends up being associated with the emission probabilities of the HMM, hence whilst suboptimal, because emission probabilities do no not define completely the HMM distribution, maximizing them helps increase the overall performances nonetheless.

Incidentally, the lower bound found for the constellation which considers

separate binary errors ends up being more fit for the particular training procedure chosen for the HMM, which is binary in nature itself. Having M different models, albeit tied on the negative state, is less efficient error wise, because the transitions between the negative and positive classes are not normalized together (i.e. $\sum_{i=1}^{M+1} \tau_{1i} \geq 1$ and $\sum_{i=1}^{M+1} \tau_{i1} \geq 1$), hence P_e found in (2.28) for the constellation is a better match for the P_e of the overall system than what could be expected by simply looking at the constellation. Better considerations on this are deemed as future work though.

3.5 Putting everything together

Now that all elements from theory to their implementation have been presented, the reasons behind the overall system can be more clearly put together: objective of this works is to create a realtime system, able to run on mid- to high-end commodity hardware, that performs continuous anomaly detection on temporal data, with particular regards to videos (coming from CCTV cameras, for example).

To tackle the continuous detection problem, input data has been divided as explained in Section 3.1. This creates a difficulty in keeping consistency among different video segments and their composing units that go through the neural network, with the risk for them to end up being treated separately and independently by the NN. The Hidden Markov Model provides then a simple solution to the problem: it assumes that each unit within a segment depends also on the previous one, linking them explicitly. This assumption though is in itself too weak to tackle the complexity of actions and consequences typically common in a video, not to mention the fact that feeding directly a video to the HMM is impossible to manage. Hence the need to train the NN and the HMM jointly with the considerations of Chapter 2 to optimize the end result as most as possible.

Lastly, it must be made clear that the space $\mathbb{Z}$ over which theory results are obtained, is not actually the space of the videos, that is instead $\mathbb{X}$. This latter data space is the input of the neural network alone, while $\mathbb{Z}$ is situated at its output, and it is this one that is of concern for the Hidden Markov Model too.

4
Methods

This chapter first presents the frameworks and hardware adopted for this work. It then introduces the datasets, then it concludes with some considerations on the training methodology.

4.1 Frameworks

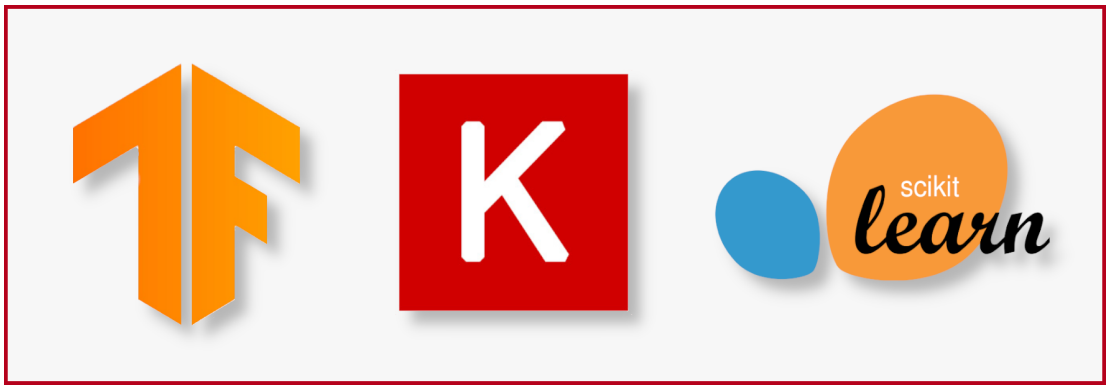


Figure 4.1: Logos of each framework: Tensorflow (left), Keras (center), hmmlearn's parent framework, ScikitLearn (right).

There are many frameworks and enabling technology for Machine Learning (ML) and Artificial Intelligence (AI). They are mostly built around the concept of computational optimization, because ML/AI workloads are notoriously heavy, with different target architectures in mind: from mobile devices to managed clusters on cloud services. User-friendliness is also important to make them available to the most widespread range of users. Tensorflow, with its Python interface and Keras abstraction layer, is chosen in this context to implement the NN because it is able to scale to every conceivable architecture, while providing an easy-to-use interface, that can be easily swapped and interchanged with the complete set of functions and tools when more control is needed. It is also a very complete and comprehensive framework, albeit still under heavy development. HMMLearn is chosen for providing an HMM implementation instead.

4.1.1 Tensorflow

Tensorflow is an open-source symbolic math library, especially useful for machine learning, that is able to operate at a large scale and in heterogeneous environments [28, 29]. Therefore, of capital importance is for it to be *flexible*.

Flexibility is achieved by means of a unified data carrier, that is a multidimensional array, called tensor. Many of these structures flow[1] around a computational graph of primitive operators, which represents the symbolic operations that the program must run. A node in such graph idealizes instead individual mathematical operators, allowing for optimizing computational performances, but also for automatic differentiation of mathematical expressions: an extremely important capability, that is leveraged by many algorithms, especially (stochastic) gradient descent. But a computational graph expresses also dependences of operators from previous ones, hence computations can be split and distributed among different hardware, and then run in parallel. Another key point in Tensorflow's architecture is the heterogeneous hardware abstraction, from which the scalability of this frameworks comes from, hence multiple hardware components among the same one or

[1] hence the name of the framework

distributed within different machines, and even made of different architectures (CPU, GPU, TPU, etc.) is handled almost seamlessly.

4.1.2 Keras

Keras [30] is a high-level neural network library, written in Python, designed with fast experimentation and easiness of use in mind, achieved in particular through modularity and extensibility. It supports a variety of backends Machine Learning libraries, such as Tensorflow, providing a uniform interface for all of them. Actually, as of the time of writing this work, Keras is officially supported and shipped within Tensorflow itself, with the role of a landing interface that enables easy prototyping of NN models.

Keras provides implementations of commonly used NN layer types, such as fully connected or convolutional, but also advanced ones, like gated recurrent layers. It also provides support for common input data types (text, images, ...) to feed the models with, but can manage also custom types, thanks to the mentioned extensibility. Lastly, it provides built-in training distribution strategies that leverage the backend's facilities to take advantage of parallel hardware.

4.1.3 HMMLearn

HMMLearn [31] is a library that boasts simple algorithms and models to learn Hidden Markov Models, and is written in Python. It follows Scikit-Learn's[2] interface philosophy, with minor adaptations to allows for sequence data tractability. It provides implementations for HMM with Gaussian or Gaussian Mixture emissions, trained with the Baum-Welch algorithm, and either the Viterbi algorithm or a dynamic programming version of a MAP algorithm, to speed up computations on sequences. It is also extensible to other distributions, provided one knows how to implement the Estimation Maximization algorithm for them. This is generally not necessary though

[2] An open-source library with simple and efficient tools for data mining and data analysis

since Gaussian Mixture are generic approximations for every distribution, provided they have enough mixture components.

4.1.4 Hardware

During the first experimentations, the models where run on commodity hardware, comprising an intel i7-8750H, an NVIDIA GTX1050Ti and 16GB of system memory. Later on, when dimensions of the NN and memory requirements to host the dataset demanded a more suitable platform to train them, development switched to a small cloud solution, with an 8-core virtual CPU, 25GB of system memory, and an NVIDIA K80 GPU. The cloud solution is mandatory only when training on the real video dataset (explained in a short while), whereas with the generated one, the commodity hardware is more than enough. Once trained, the models were tested for realtime performances on a desktop PC equipped with a GTX 1070, an Intel Core i7-6700K, and 16GB of RAM.

4.2 The datasets

A few datasets were used in this work. One is generated starting from fashion-MNIST and it is intended to test the general behavior of the system, especially against what has been uncovered in Chapter 2. Then UCSD anomaly detection dataset was briefly used but found heavily inadequate, and later replaced by UCF Crime, although this ended up not resolving the lack of a sizeable ensemble of training samples either. Each dataset is described in detail in what follows.

4.2.1 Generated dataset

Fashion-MNIST [32] is a dataset of Zalando's article images, that keeps the same size and proportion of the original Digit-MNIST, that is 60,000 training samples and 10,000 testing samples, divided equally among 10 classes. Each sample is constituted by a 28x28 image, of grayscale intensities and with

the background already removed. The dataset is intended as a more complex benchmarking dataset than Digit-MNIST, and the classes are T-Shirt/Top, Trousers, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, bag, Ankle boot, labeled from 0 through 9.

However it is made up of still images, hence there is the need to turn it into a video sequence for this work. The idea is to pile images sequentially to make a video, of course. The frames per second are fixed at 15, to match the output of data augmentation on the two real datasets. One of the classes is elected as the normal class (the 0 labeled T-Shirt/Top class), and it is given the highest probability of appearing. All other classes are threated as anomalies and they are given an equal probability of appearing in the video, which is strictly less than that of the negative class. To actually generate a stream of labels (out of which later create the video) that complies to Assumption 1, a 10-states Markov Chain with a handcrafted transition matrix that has nonzero values only on the main diagonal and both the very first row and column, is sampled a sufficient number of times. For each consecutive stream of the same label, a random image is extracted from the respective class to compose the final video either from the train or from the test pools of samples, depending on the circumstances of the training procedure. Random rotations were also applied to images but quickly removed too, because the very low resolution made the images quickly fade to a uniform blob.

4.2.2 UCSD Anomaly Detection dataset

Figure 4.2: Three snapshots from UCSD Anomaly Detection dataset, showcasing three different anomalies (highlighted within boxes), and the two available scene backgrounds.

The UCSD Anomaly Detection is a dataset collected on the campus of the

University of California San Diego by means of a stationary camera overlooking two pedestrian walkways.

Anomalies, as considered by the authors, are:

- the circulation of non pedestrian entities in the walkways
- anomalous pedestrian motion pattern

Commonly occurring anomalies include bikers, skaters, small carts, people in a wheelchair, and people walking across the grass rather than staying on the walkway. All events were not staged, as they were captured as naturally occurring [33]. Each video features one of two possible scenarios as visible in Figure 4.2, with different walking directions of people. Lastly, clip length is around 150 frames, each with dimensions 238x158 pixels for the first scenery and 360x240 pixels for the second. The first scenery also has a total of 34 clips void of anomalies and 36 with at least one anomaly, wheres the second scenery has 16 and 12 respectively.

This dataset quickly became inadequate, due to too few clips to train a neural network on direct anomaly detection when the anomaly itself is so small compared to the frame (notice the relative size of the red boxes in Figure 4.2 compared to the whole frame). Indeed the only information the system is provided with is whether or not there is an anomaly in a group of frames, but not where such an event is located spatially to help the NN in learning useful features. There have been attempts instead at a more indirect approach, based on the reconstruction of the normal scene to find the anomaly by difference, like in [34] where a complex system of autoencoders and a generative adversarial network is used with such end, which can achieve some success in this dataset.

4.2.3 UCF Crime dataset

UCF Crime is a dataset consisting of 1900 untrimmed surveillance videos grouped in 13 classes of risks, plus a normal class, selected because of their *impact on public safety* [3]. Videos were collected from YouTube and LiveLeak using queries holding the classes as keywords within, in different languages

Figure 4.3: Some snapshots from UCF Crime dataset. Difficulties inherent this dataset are also visible: events are little compared to frame size and often out of focus.

as well. The dataset contains videos coming from CCTV cameras, but many instances where captured as part of some news, hence there are overlays and cuts, as well as many repetitions of the same scene from different perspectives. The vast majority of videos are colored, but there are some that are in grayscale only. Lastly, each video was elaborated to account exactly 30 frames per second, and have 320x240 (4:3) resolution.

Only video level labels are provided with the dataset: manual annotation of the per-frame ground truth was carried out for this work only on the "Road Accidents" class. The criterion used is: the anomaly starts at the first moment two or more vehicles touch each other, or when a vehicle first touches one or more obstacles/people and ends the first instant all the involved parties lay still. Each anomaly thus highlighted lasts less than 10 seconds with a mean of $2.5s$. To avoid repetitions within the same video, portions of 15 seconds are cut from these videos, each having an entire abnormal sequence plus some contours, to reach the target temporal span, where the situation is still considered as normal. The proportion of negative and positive labeled units is 83%

and 17% respectively.

This dataset proved itself very challenging, up to the point that it was impossible to extract meaningful results out of it. Part of the difficulty comes from the same reasons explained for the UCSD dataset: anomalies are small compared to the whole frame size, and because the task is to classify the anomalies other than detecting them, there are unfortunately too few samples per class to leverage for training. This problem is made worse by the fact that situations within the same class also boast high variance. For instance, people hit by cars, bikes falling for many reasons, cars hitting urban obstacles, cars hitting other vehicles, all with highly varying degrees of ambient conditions ranging from sunny days to snowy ones going through night conditions as well, are all classified as road accidents.

CLASS	# OF VIDEOS
Abuse	50
Arrest	50
Arson	50
Assault	50
Burglary	100
Explosion	50
Fighting	50
Road Accidents	150
Robbery	150
Shooting	50
Shoplifting	150
Stealing	100
Vandalism	50
Normal	950

Table 4.1: Classes within the UCF Crime dataset and their size. Table reproduced from [3].

Data augmentation was also applied to this dataset. Firstly, to double the clips, the time dimension was sampled in an interleaved fashion, obtaining two videos out of the original one, but each with 15 instead of 30 FPS. More aggressive sampling could have been tried, but this would have reduced the number of time samples per second too much while generating too similar samples as well. Whole videos were also randomly flipped along the horizontal dimension (the vertical one wouldn't make sense) and a small amount of

white noise or blur was added randomly. The common technique of region cropping to obtain more clips could not be applied, due to the mentioned small relative size of the abnormal region withing each frame.

4.3 Training method

The last things left to discuss in this chapter are some considerations on the training procedure.

Training is done in a fully supervised manner, with manually annotated labels. This means also that this method relies on a good number of examples coming from the positive classes, an assumption that is not always possible to adhere to in the context of anomaly detection. Further work must concentrate therefore in two directions: (i) relax the supervised approach, with a more weakly oriented one, maybe leveraging a framework such as Multiple Instance Learning [35, 36]; this means to provide only video level labels, as the UCF Crime dataset already does, instead of frame-wise or even pixel-wise labels; (ii) consider if it is possible to adapt or extend this method to learn a general definition of normal behavior, to then understand that there is an anomaly simply by noting that the analyzed signal behaves differently by what has been experienced during training; ideally, this way the system would be able to classify all the positive instances that had enough samples during the training phase, and for all those anomalies that didn't, the system would still be able to say there is a generic abnormal situation happening.

Moving on the data-feeding pipeline instead, the Neural Network receives its inputs, regardless of which dataset they come from, in grayscale format (if they aren't already), and in batches of consecutive video segments (of size 32 when using generated videos, and size 20 when using real videos, due to memory constraints), rather than one at a time; each batch is also standardized: the mean μ_B and the standard deviation σ_B are found, and then each frame $\mathbf{x}$ of each segment undergoes the following transformation:

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu_B}{\sigma_B}$$

The NN is then fed with $\hat{\mathbf{x}}$. No other transformations are applied to the input, following a data-driven[3] training paradigma.

When it comes to Algorithm 1 instead, it has to be said that at each iteration k the NN must be trained to a sufficient level of accuracy for the current configuration of the target constellation. This means that for each k, the NN is trained a certain number of epochs, currently fixed at 10, and the learning rate of the gradient descent optimizer is also reduced by some schedule depending on k. Then the HMM can be trained. It also has to achieve some target level of accuracy in its estimations of the gaussian constellation: currently, the tolerance for the EM algorithm is set to an error of 1×10^{-3}. Lastly, after each iteration k, the system also undergoes a validation test to see how it is performing overall.

To conclude, as mentioned in Chapter 3, some important modifications to the architecture were also experimented with: (i) Auto Encoders and (ii) transfer leaning from pre-trained NN models.

An Auto Encoder (AE) is a type of neural network that aims at reconstructing its own input, that is, it aims to perform an identity function. This would be useless, if it wasn't for the fact that in doing so the NN is also forced to encode all the information needed for the reconstruction of the input in as little space as possible, hence there always is a choke point in the architecture of an AE. Such choke point is in practice a feature vector, therefore AEs are often used for their ability to extract meaningful and compressed features. This is possible because the first part of the AE structure, before the choke point, learns to look only at relevant details in the input, while the second part of the structure, after the feature vector, learns to encode all the redundant and correlated portions of the input, to later add such information during the reconstruction of the details from the feature vector, and therefore obtain an output as close as possible (by some chosen metric) to the input.

While describing the datasets, it was noted that the abnormal event in a video occupies only a small portion of the frame. The rationale beyond using an AE in the system architecture, therefore, is to have the second half of the AE

[3]The NN must learn everything it needs from the original data, without external conditioning, for instance by providing handcrafted features as inputs alongside or in substitution of the videos.

learning all the useless portions of the video signals, allowing the first half to better concentrate on the relevant patterns within each video segment. After the training phase, the second part of the AE can be disposed of, obtaining also a more compute-efficient system (than the whole AE) as a byproduct.

It was also noted that the datasets have too few samples to work upon. Transfer learning is therefore intended to jump-start the gradient descent for the NN from a solid starting point, the result of a previous training procedure, that is hopefully closer to the global optimum rather than a casual starting point obtained from a random initialization of the weights of the NN.

A discussion about what results these two architectural modifications brought is left for the next Chapter, though.

5 Results

THIS CHAPTER AIMS to understand what kind of results the proposed algorithm and its implementations yields. Therefore the binary class case with the generated dataset is analyzed firstly, to provide a baseline for later implementation and dataset variants. Then, the multi-class instance, with the generated dataset as well, is put under the magnifying glass, followed by the binary class realization of the auto-encoder and the I3D architecture on the UCF crime dataset. UCSD's dataset is not taken into consideration because of already mentioned problems.

The properties analyzed for each implementation are twofold:

1. how well a method adheres to results from Theorem 2.
2. how good of a classifier the method is.

The metrics tracked for the first point are the neural network's loss over training and validation epochs[1], the HMM's log likelihood over iterations, and, most importantly, the error plot between the target distribution P_{true} of the $\mathbb{Z}$ space, the NN's output distribution P_{NN}, and the HMM's distribution P_{HMM}. In such plot P_{true} is also overlayed on top (or beside if more convenient) to make clear what is the target distribution's aspect. The plot actually displays

[1]Each iteration k of Algorithm 1 comprises 10 epochs for the NN in the current setup

the MAE error function for each of the L sampled points in $\mathbb{Z}$, computed as follow:

$$MAE(z) = |2P_{true}(z) - P_{NN}(z) - P_{HMM}(z)| \tag{5.1}$$

Because P_{NN} can only be estimated by computing the samples' histogram, Confidence Intervals for $\alpha = 95\%$ are shown too. The only exception is the plot for the multi-class case, where for graphical reasons first, Confidence Intervals are omitted, and second, the actual 3D constellation is projected onto two dimensions only to allow for the plot to be drawn.

Lastly the Mean Absolute Error is estimated as:

$$MAE = \sum_{l=1}^{L} \frac{MAE(z_l)}{L} \tag{5.2}$$

For the second point, instead, the confusion matrix is computed, also in its normalized version. Such matrix encodes information on how well each class is identified correctly, as well as displaying both for which classes the current one is mistaken (type I error, false positives) and which classes are mistaken for the current one (type II error, false negatives). Out of this matrix, precision and recall scores are computed as well to better understand how type I and II errors affect the system, and the overall harmonic mean of these two, called F_β-score, is computed too as

$$F_\beta = (1 + \beta^2)\frac{precision \cdot recall}{\beta^2 \cdot precision^{-1} + recall^{-1}} \tag{5.3}$$

The parameter β weights the precision against the recall, and $0 \leq F_\beta \leq 1$ (the closer to 1 the better). Since at the moment there is no clear need to optimize the system's performances to minimize either type I or II errors, β is set to 1.

It should also be noticed that the worst-case classification scenario is adopted: it is not sufficient for the system to just identify a portion of an anomalous sequence to say that a correct detection happened; it must instead identify individual anomalous units within a video.

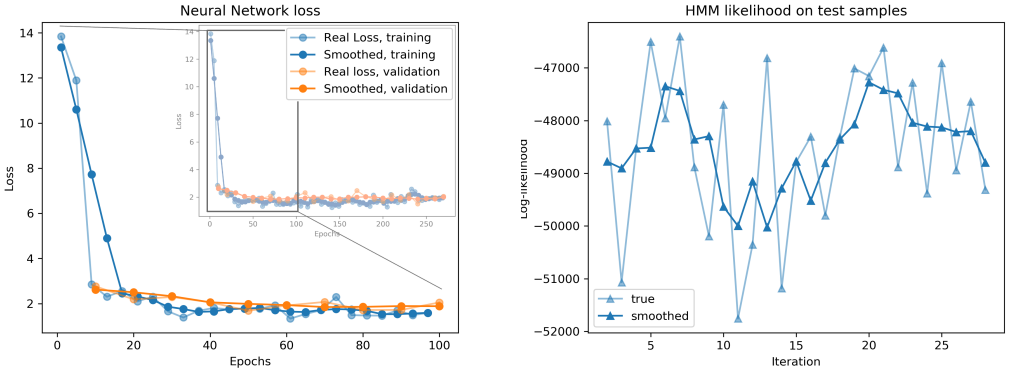
5.1 Still open problems

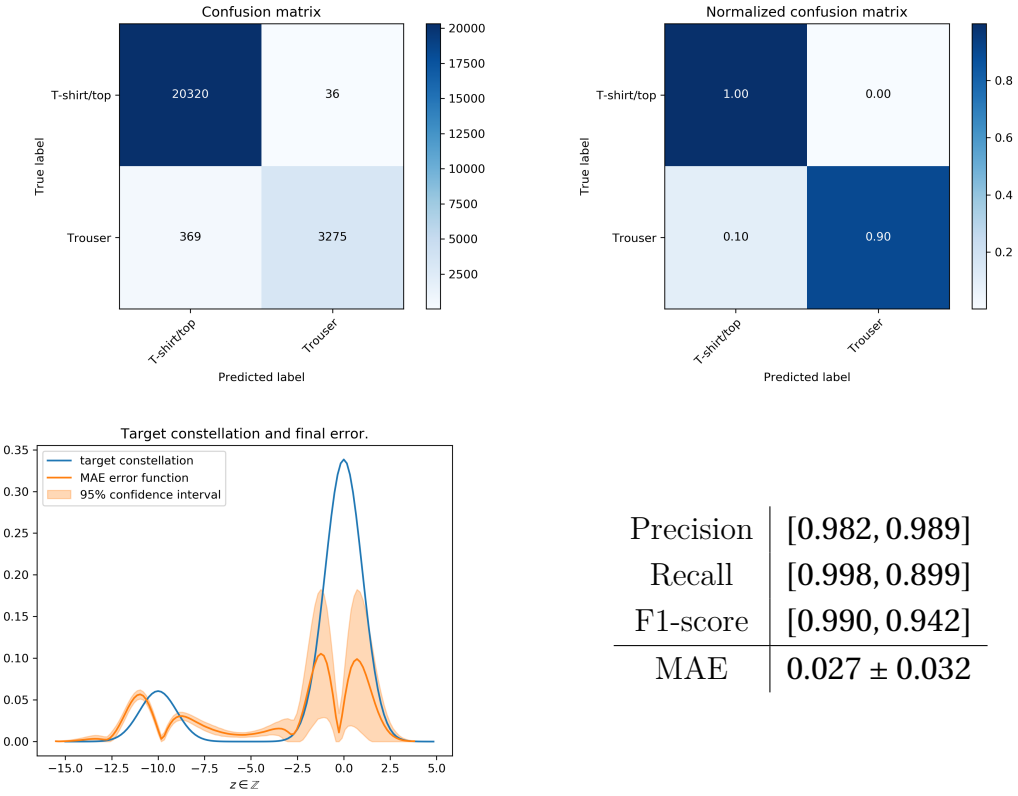
There are some open problems that remains to be solved:

- **Stopping condition**: in Algorithm 1 the stopping condition is attained when the series of $g_k(\cdot)$ functions reaches the identity. This is however difficult to obtain in practice, because of both the NN's and HMM's inability to reach a perfect optimum. The problem is that without such condition the NN can be over-trained, worsening the results. Common techniques to stop an NN from over-train can be adopted, but more studies need to be carried out to understand the impact, if any, on the overall method.

- **Choosing the best HMM**: in the multi-class instance, the best HMM is selected as per the criterion explained in Section 3.4. This may not be the optimal choice, therefore some more considerations into this matter can prove useful, in the future.

5.2 Generated dataset: binary case





Precision	[0.982, 0.989]
Recall	[0.998, 0.899]
F1-score	[0.990, 0.942]
MAE	0.027 ± 0.032

Figure 5.1: Results obtained without using class weighting (Generated dataset, binary case).

The binary classification task with the generated dataset behaves mostly as expected. In Figure 5.1 The neural network reaches a good value for the training loss quickly in a bunch of epochs, and later, as it continues to improve slowly, the validation loss sticks close to that of the training and shows an overall good behavior. Fluctuations in both losses are due to the constant changes of images used to build videos. The HMM likelihood oscillates more wildly, but it must be considered that it is the end sum of thousands of sequence's likelihoods (actually 4200 5-units segments), hence even small oscillations do matter a lot. No general trend can be spotted for such curve, however, mostly because the NN converges quickly to it's optimum and therefore the overall system benefits from this, stabilizing itself around its optimum too. This also means that so many iterations and epochs are not strictly necessary, hence the need for a good stopping condition. It has to be noted also that the very first HMM's likelihood was removed from the plot, since it has been executed

against the random output from the NN, and therefore it has no actual meaning.

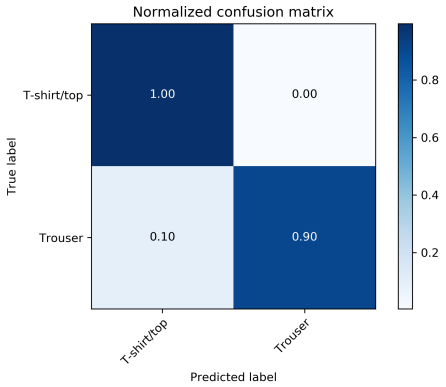
The good behavior of the system is matched by the (normalized) confusion matrix, where it can be seen that the negative class is recognized almost perfectly, whereas the positive class, although it exhibits some type II errors (false negatives), is found correctly 90% of the times. The F1 score confirms this, being close to 1 for both classes.

The shape[2] of the error function (5.3) reveals the culprit for such inefficiencies: it is mostly the NN that is unable to match the target Gaussian constellation, since it shows indecision on some samples, that ultimately causes for it to put them somewhere in the middle between the two classes. This is the root cause of the error around the values in the interval $[-7.5, 2.5]$ in the plot. In turn, this indecision causes the HMM to try to match the NN's output distribution, pumping up the double camel-like hills around -10 and 0, more than what the sole NN would have caused. However two things should be noted: (i) the error near -10 and 0, the two chosen constellation centers, is close to zero, hence a lot of samples do end up having the right distribution, and (ii) the overall MAE is low.

Still, future work may want to concentrate around solving such NN inefficiency, improving its training routine, while at the same time it may consider changing the HMM's emission distribution to the more general Gaussian Mixture, and the target distribution too, in an effort to reduce the described effects. Moreover, Theorem 1 is instead well adhered to, the problem being mostly in some difficulty in converging towards the identity function as forecasted by Theorem 2.

Lastly, Figure 5.2 shows the most relevant results when class weighting is applied. It must indeed be remembered that there is a heavy class unbalance, as can be seen from the unnormalized confusion matrix. Therefore in computing the loss for the NN, the positive class contribution was boosted until it matched the negative class's. This showed, however, to bring no benefit at all in this instance, mostly because of the simplicity of the generated video.

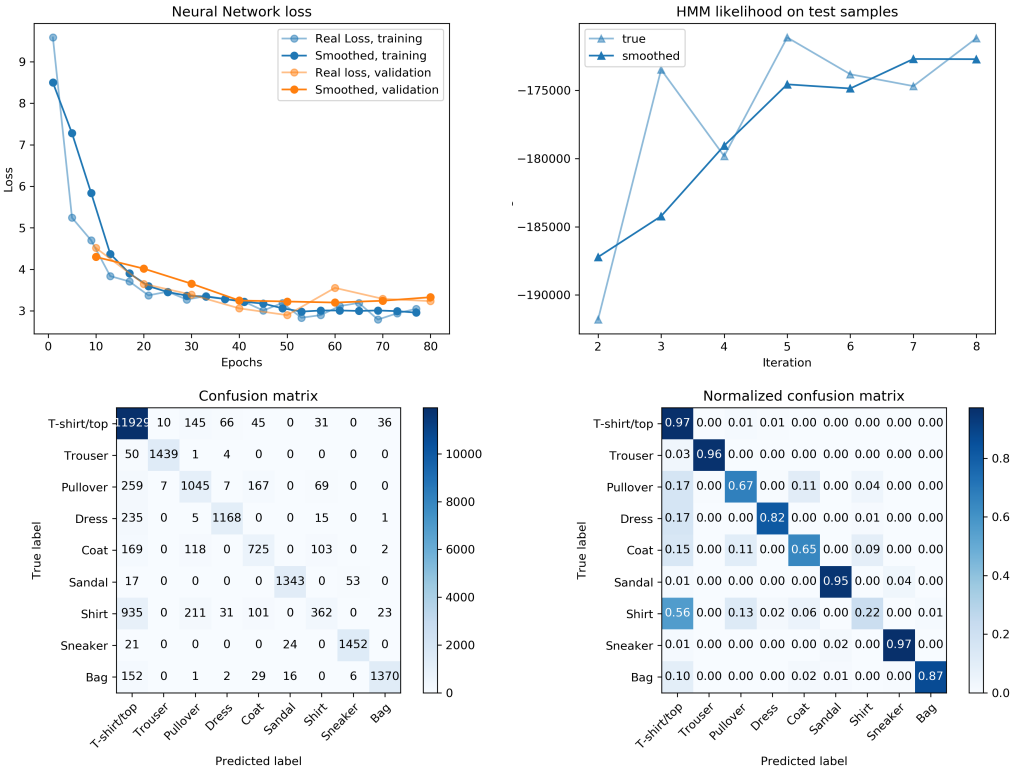
[2] the closest to zero the better fo course

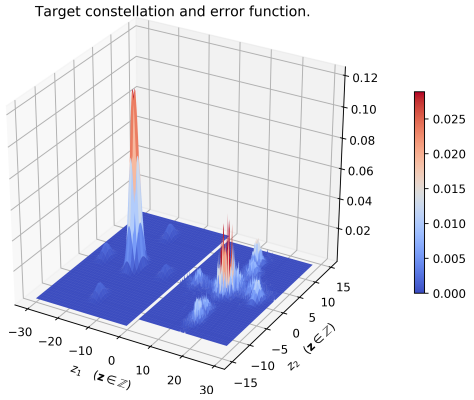


Precision	$[0.983, 0.975]$
Recall	$[0.996, 0.905]$
F1-score	$[0.990, 0.938]$
MAE	0.028 ± 0.030

Figure 5.2: Relevant results obtained using class weighting (Generated dataset, binary case).

5.3 Generated dataset: multiclass case





Precision	[0.87, 0.99, 0.68, 0.91, 0.68, 0.97, 0.62, 0.96, 0.96]
Recall	[0.97, 0.96, 0.67, 0.82, 0.65, 0.95, 0.22, 0.97, 0.87]
F1-score	[0.92, 0.98, 0.68, 0.86, 0.66, 0.96, 0.32, 0.97, 0.91]
MAE	0.001 ± 0.003

Figure 5.2: Results obtained without using class weighting (Generated dataset, multi-class case).

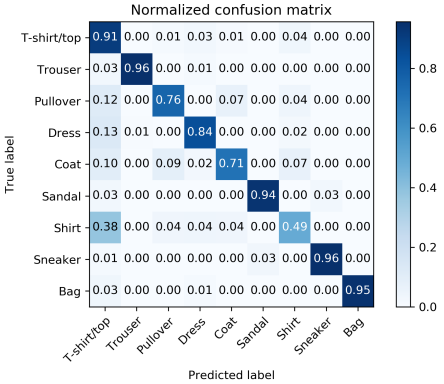
Figure 5.2 shows the results for the multi-class instance instead. As emerged from trials, and as can be seen anyway slightly towards the end of the NN's validation loss (that has a slight tendency in diverging from that of training), here there is the risk for overfitting, hence the system was stopped training manually to avoid this phenomenon. Instead, the HMM's likelihood exhibits the expected behavior: it improves with the iterations. There are actually no guarantees on how the likelihood must behave, but it is expected to improve as the NN's output distribution becomes closer to the gaussian constellation, because this was chosen since it can be better represented by the HMM rather than more unstructured ones.

The classification performances are overall good, as told by the (normalized) confusion matrix. Some classes are more difficult for the NN to classify into the correct peak of the constellation, hence some error is present for the overall system, for the same reasons exposed in the earlier section. As expected though, mostly such positive classes are misunderstood for the negative one, because they are well under-represented with respect to the latter. This is reflected by the precision and recall metrics, with the first being consistently higher than the second. The errors are understandable as well: the most misplaced class is the "SHIRT", confused with the much more frequent, and very similar indeed, "T-SHIRT" (negative) class.

For graphical reasons, the error function is plotted only in two dimensions rather than the total 3 (there are $M + 1 = 8 + 1$ classes in total, which gives 3 dimensions for the constellation), and for the same reason Confidence In-

tervals could not be drawn. The same NN's indecision can, however, be spotted again between the smaller peaks, belonging to positive classes, and the central highest peak, representing the normal class. More in general, the behavior is the same as for the binary case, only in more dimensions. MAE is again low, actually much lower than before, but this is also the result of having more dimensions to account for, hence performances are the same as the binary instance.

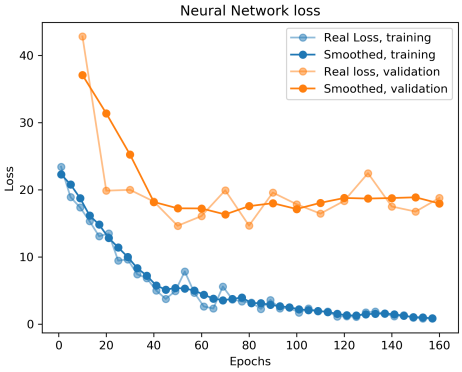
Lastly, class weighting, as visible in Figure 5.3 brought some benefit, increasing reconnaissance in those classes that struggled more before, penalizing slightly the normal class instead.

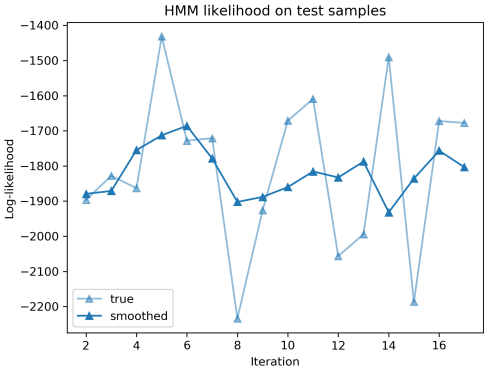


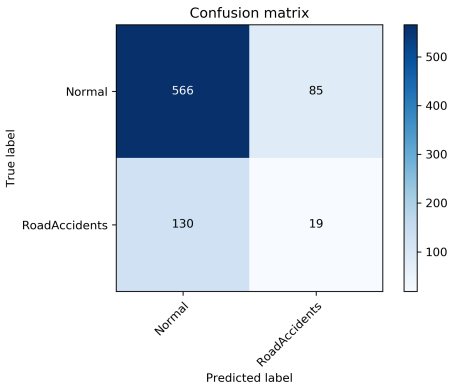
Precision	[0.91, 0.96, 0.81, 0.71, 0.75, 0.96, 0.51, 0.97, 0.97]
Recall	[0.91, 0.96, 0.76, 0.84, 0.71, 0.94, 0.49, 0.96, 0.95]
F1-score	[0.91, 0.96, 0.78, 0.77, 0.73, 0.95, 0.50, 0.96, 0.96]
MAE	0.001 ± 0.006

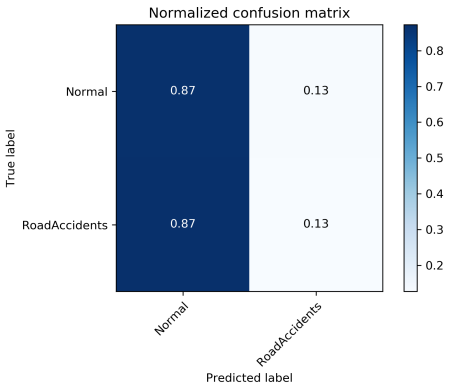
Figure 5.3: Relevant results obtained using class weighting (Generated dataset, multi-class case).

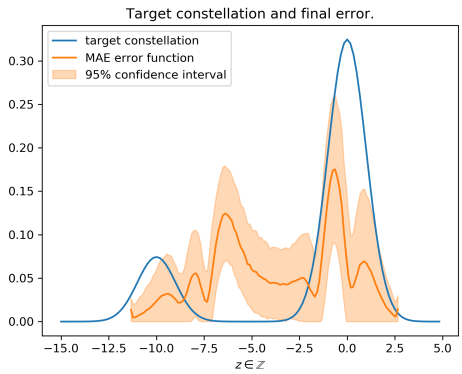
5.4 UCF crime: auto-encoder











Precision	[0.813, 0.183]
Recall	[0.869, 0.128]
F1-score	[0.840, 0.150]
MAE	0.055 ± 0.047

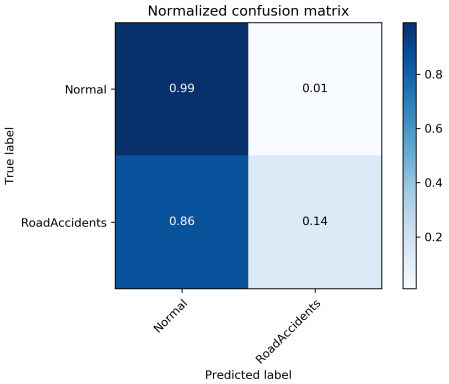
Figure 5.4: Results obtained without using class weighting (UCF crime, auto-encoder).

Switching to the UCF Crime datasets caused a whole lot of troubles, and to try to solve them this and the following methods were attempted. It can be seen immediately in Figure 5.4 however that the NN training is not successful: while the training loss keeps decreasing, the validation loss either stays constant or tends to increase rather than doing the converse, and this means that the network is seemingly overfitting on the training data, whilst being unable to generalize, even to new data coming from the same (unknown) generating process. And the auto-encoder structure is of no help in this circumstance since the decoder part of the NN is not able to guide the encoder towards discriminating those relevant volumes of videos that are of interest for the purpose of anomaly detection. This is probably due to the same reason as a more straightforward method didn't work either: the inability to find the relatively small volume that encloses the anomaly within the video segment, mostly because of a lack of sufficient data.

Class unbalance, that should have made the NN gravitate towards identifying everything as the normal class, since this is an easy way to optimize the loss in this circumstance, being this class over-represented, instead did not play a role. This is visible both in the (normalized) confusion matrix, where a surprisingly strong classification ability[3] for the positive class is present, and in the error function that shows errors are widespread and consistent along most of the relevant portion of the $\mathbb{Z}$ space. As for the misclassification error at 13% committed for the normal class, or the same for the positive class (at 87% instead) it should be noticed that relative class frequencies are exactly 87% and 13%, therefore because the error is widespread, this means basically that the NN is actually guessing mostly randomly the mean and variance to assign to each unit of a video, rather than overfitting to the normal class only, therefore units end up being classified either correctly (the diagonal blocks in the confusion matrix) or wrongly (the anti-diagonal blocks) as positive or negative by the HMM with the same chances as the classes actually appear in the dataset. Quite a different effect should have happened if the NN would have overfitted to the normal class only. This shows that the AE structure has some regularization capability, that may be of interest to studying on a more adequate dataset.

[3]The ability is very poor at 13% only, but it is relatively strong as it is totally unexpected.

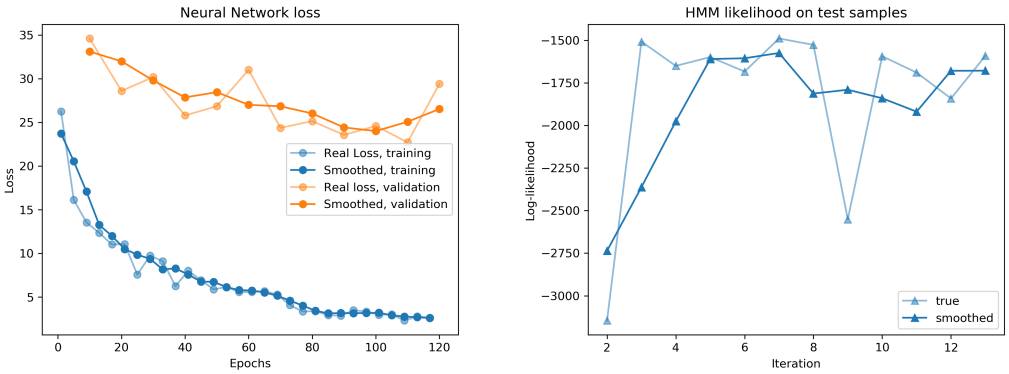
Introducing class weighting instead strangely favors the negative class, whilst the positive class has no appreciable benefits whatsoever, as Figure 5.5 shows. This is probably because now the positive class is slightly more decisively discriminated in the few instances the NN is able to do that, and this is sufficient for the HMM to correctly classify the normal class also in those 13% situations that previously were ambiguous. The converse is not true instead, leaving the misclassification error fro the positive class high, at 86%.

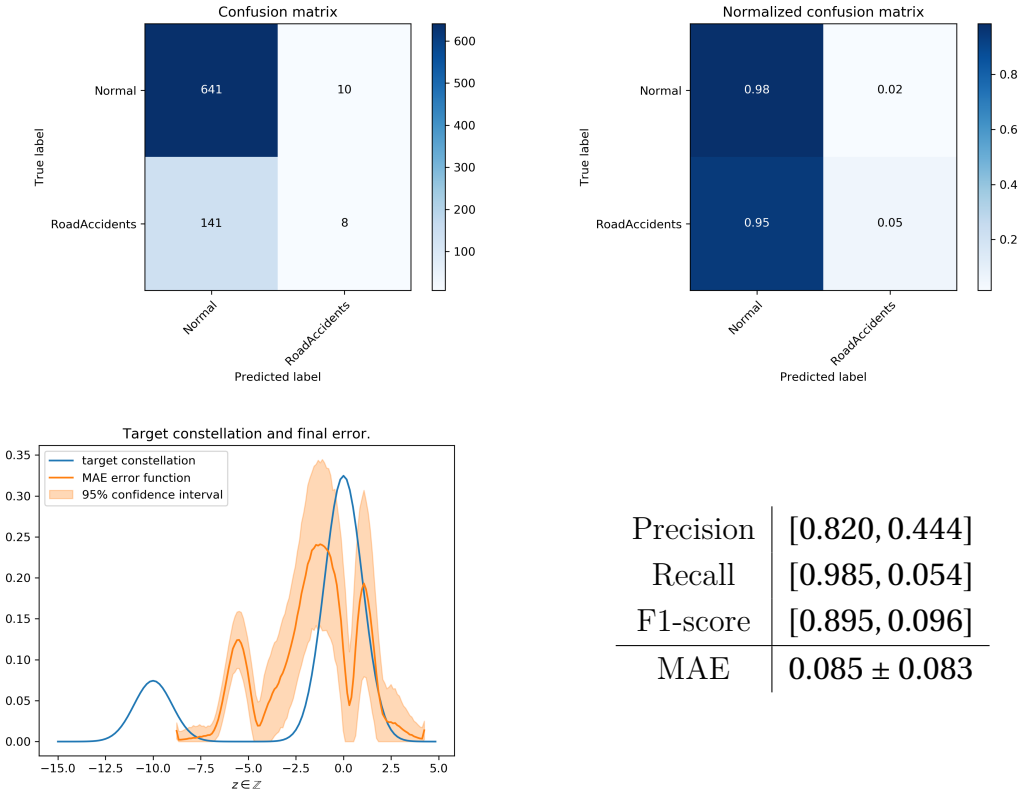


Precision	$[0.834, 0.778]$
Recall	$[0.991, 0.141]$
F1-score	$[0.906, 0.239]$
MAE	0.056 ± 0.043

Figure 5.5: Relevant results obtained using class weighting (UCF crime, auto-encoder).

5.5 UCF crime: transfer learning from I3D





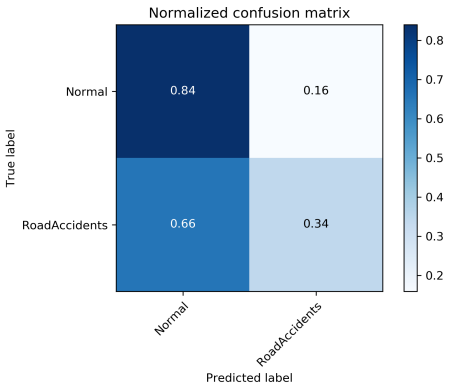
Precision	[0.820, 0.444]
Recall	[0.985, 0.054]
F1-score	[0.895, 0.096]
MAE	0.085 ± 0.083

Figure 5.6: Results obtained without using class weighting (UCF crime, I3D).

The situation shown in Figure 5.6 is similar to that of the previous method when it comes to the NN and HMM training. But now there is actually a strong tendency of overfitting towards the negative class, with a (normalized) confusion matrix that shows that pretty much every video unit is though to be normal by the HMM. The error function indeed is more concentrated towards the negative class this time. The cause of this can be sought in the initialization state for the NN: it was trained on the Kinetics dataset [37], a rather different one in kind from UCF, and this means that such initialization point is basically equivalent to a random one from the perspective of this problem. Unfortunately, it was impossible to find a suitable model trained on a similar dataset as for the needs of this work.

This, jointly with the lack of samples made the NN unable to find even the slighted way to discriminate positive and negative classes from each other, at least without providing class weighting, as it seems from Figure 5.7. Indeed

there is a dramatic improvement in the latter instance when it comes to the positive class, albeit not nearly as good as is desirable. This seems to be a promising direction, but the training procedure must be revised in order to try to obtain some relevant results, because, again, the system was manually stopped as it was gradually overfitting and going towards the same results as in the no-weights case: it simply was taking longer to do that.



Precision	[0.848, 0.329]
Recall	[0.840, 0.342]
F1-score	[0.844, 0.336]
MAE	0.096 ± 0.127

Figure 5.7: Relevant results obtained using class weighting (UCF crime, I3D).

5.6 Real time performances

A supporting data ingestion pipeline is not within the aims of this work, hence it is assumed that upstream on a production system there is a data source able to provide videos at 15 or 30 FPS in realtime from CCTV cameras. With this premise, the system runtime performances were tested on a desktop PC equipped with a GTX 1070, an Intel Core i7-6700K, and 16GB of RAM.

The bottleneck is represented by the Neural Network which is computationally expensive, and since it runs off the GPU, it requires data transfer back and forth from the main system memory (RAM) to the VRAM on the GPU, a notoriously slow operation, which is however optimized internally by TensorFlow. The separate HMM models for the different classes are small and computationally very fast instead, moreover, they can be executed in parallel to each other on the output of the NN.

MODEL	RUNTIME (seconds)	FPS
AE	$8.66 \pm 0,06$	~ 277
I3D	17.33 ± 0.21	~ 138

Table 5.1: Runtime performances of the AE and I3D models.

The models tested were for the AE and the Transfer Learning cases. The first has about 35M parameters in a ResNet architecture, of which only about 20M active during evaluation (since the others belong to the decoder, not needed aside when training the model), whereas the second has 12.5M, in an Inception 3D architecture. The models were repeatedly asked to analyze 2400 frames in total (that is 160 seconds on video), obtaining the following:

The reason for the I3D model performing worse, albeit smaller, is due to its multiple-paths architecture, which creates many waiting points in TensorFlow's computational graph, compared to the more linear architecture of a ResNet. It can be seen in Table 5.1 that both models are more than able to cope with the required 15 FPS or 30 FPS of the supposed video stream.

6

Conclusions

THERE ARE MANY WAYS, SYSTEMS, IDEAS, on how to perform anomaly detection since the concept of anomaly itself is ambiguous and especially is context-dependent. Therefore most often than not, handcrafted solutions are required, depending on each task, as a universally good performing anomaly detector doesn't exist.

Hence, this work aims to find an anomaly detection and classification method suitable for video analysis. Starting therefore from the definition of what is an anomaly in this context, a theoretical framework that satisfies the requirements behind this work has been presented and discussed in detail. This thesis also introduced an architecture that can implement such a framework, based on a combination of a Neural Network and a Hidden Markov Model.

The architecture, albeit not perfect, behaves mostly as expected when it comes to adhering to theoretical results. A practical application couldn't be pursued unfortunately instead, due to technical difficulties related to the dataset, and more work is required around this problem.

6.1 Future work

Future reflection points have been already proposed throughout this thesis. To sum them up, the most important future direction to pursue is to stabilize the system to work properly in a real situation. This not necessarily implies finding a better dataset, since UCF already is a good starting point (albeit it can enjoy some expansions), but rather a better NN architecture, able to correctly identify small portions of video volumes as the focal point where an anomaly is happening. Some sort of attention mechanism perhaps based on [7] may be sough to this end, as well as switching to a (gated) recurrent structure. Coupled with this, further insights into the Auto-Encoder architecture may be of interest as well.

After this has been accomplished, to implement a weakly supervised training procedure is of most relevance. A candidate framework has been found in the Multiple Instance Learning paradigm [35], also because it showed already some results on the UCF Crime dataset itself, as described by the researchers that created it.

Of a slightly less priority, there is the need for fine-tuning the training procedure for the system, by solving the two points exposed in the previous chapter, namely a better stopping condition to avoid over-training and a more consistent way of choosing the best performing HMM among the pool when multiple positive classes are to be discriminated.

From a theoretical standpoint, switching to a different emission probability for each HMM state must be evaluated, with particular reference to the Gaussian Mixture, which has established proofs in the literature of improving the HMM model performance. Also, a better understanding of the coupled NN-HMM system error, and how it is influenced by individual errors of the two models is required. In particular, it has been seen in the previous chapter that critical for the performance is still the good behavior of the NN, more than the HMM's.

To conclude, anomaly detection introduces many interesting challenges, keeping the research interest alive and stimulated, as well as providing countless possibilities for constant improvement.

A

Additional Theory Notions

THIS APPENDIX contains some material taken from other sources that comprise useful notion for Chapter 2, reported in this work also for the sake of completeness.

A.1 Converging to Equilibrium

Proposition 1 cites the following result from [22, Proposition 2]: the Generative Adversarial Networks model optimizes the following function:

$$V(G,D) = \mathbb{E}_{x\sim p_{data}}[logD_G(x)] + \mathbb{E}_{x\sim p_G}[log(1-D_G(x))] \tag{A.1}$$

and for any fixed G (and hence p_G), D optimizes the virtual training criterion

$$U(p_G,D) = \max_D V(G,D) \tag{A.2}$$

So then:

Proposition 2 *If G and D have enough capacity, and at each step of Algorithm 1, the discriminator is allowed to reach its optimum given G, and p_G is updated*

so as to improve the criterion

$$V(G,D) = \mathbb{E}_{x\sim p_{data}}[logD^*_G(x)] + \mathbb{E}_{x\sim p_G}[log(1 - D^*_G(x))] \tag{A.3}$$

then p_G converges to p_{data}.

Proof.
Consider $V(G,D) = U(p_G,D)$ as a function of p_G as done in the above criterion. Note that $U(p_G,D)$ is convex in p_G. The subderivatives of a supremum of convex functions include the derivative of the function at the point where the maximum is attained. In other words, if $f(x) = sup_{\alpha\in\mathcal{A}} f_\alpha(x)$ and $f_\alpha(x)$ is convex in x for every α, then $\partial f_\beta(x) \in \partial f$ if $\beta = argsup_{\alpha\in\mathcal{A}} f_\alpha(x)$. This is equivalent to computing a gradient descent update for p_G at the optimal D given the corresponding G. $sup_D U(p_G,D)$ is convex in p_G with a unique global optima [as proven in [22, Theorem 1]], therefore with sufficiently small updates of p_G, p_G converges to p_x, concluding the proof.

□

A.2 Game Theory Bits

What follows is taken from [38, chapters from 7 to 10].

First, let's introduce the concept of strategy in a game with $i = 1, ..., n$ players, where each of them, when it's their turn to move, has some knowledge, formally described by information sets $h_i \in H_i$, about the state of the game and other players' strategies:

Definition 1 *A pure strategy for player i is a mapping $s_i : H_i \to A_i$ that assigns an action $s_i(h_i) \in A_i(h_i)$ for every information set $h_i \in H_i$. A mixed strategy σ_i is just a probability distribution over all pure strategies of player i.*

Then, in a game, each player has a payoff (a score on how it values the particular outcome of the game) $v_i(\sigma_1, ..., \sigma_i, ..., \sigma_n)$ that depends on his strategies, but also the strategies of all other players. At this point:

Definition 2 *The mixed strategy profile* $\sigma^* = (\sigma_1^*, ..., \sigma_n^*)$ *is a Nash equilibrium if for each player* σ_i^* *is a best response to all* σ_j^*, $j \neq i$, *that is*

$$v_i(\sigma_1^*, ..., \sigma_i^*, ..., \sigma_n^*) \geq v_i(\sigma_1^*, ..., \sigma_i, ..., \sigma_n^*) \quad \forall i, \sigma_i \tag{A.4}$$

In a rational setting as assumed in Game Theory, all players aim at maximizing their payoff, whereas cooperation is not guaranteed unless particular conditions are present. The Nash equilibrium represents just an equilibrium point where all players are guaranteed to obtain maximum payoff in *relation to other players' strategies*, but it doesn't entice cooperation on its own since it arises from selfish considerations of rational payers. It may be the case that cooperating assures an overall higher payoff for each player, but further deepening these considerations is out of scope for this work.

Lastly, the Nash equilibrium is extended as follow in the context of repeated games:

Definition 3 *Let G be an (extensive-form)* n*-players game. A strategy profile* $\sigma^* = (\sigma_1^*, ..., \sigma_n^*)$ *is a subgame perfect Nash equilibrium if for every proper subgame G' of G, the restriction of* σ^* *to G' is still a Nash equilibrium in G'.*

hence, finally:

Proposition 3 *Let* $G(\delta)$ *be an infinitely repeated game with* $i = 1, ..., n$ *players, and let* $(\sigma_1^*, ..., \sigma_n^*)$ *be a static Nash equilibrium strategy profile of the stage game G. Define the repeated game strategy for each player* i *to be the history independent Nash strategy* $\sigma_i^*(h) = \sigma_i$ *for all history* $h \in H$ *(that is, each player always plays its Nash equilibrium strategy* σ_i *at every turn). Then* $(\sigma_1^*, ..., \sigma_n^*)$ *is a subgame-perfect equilibrium in the repeated game for any discount* $0 < \delta < 1$.

The proof is omitted. This proposition simply says that if players start to play with a Nash Equilibrium strategy for the first stage game, they'll have to keep playing it ad infinitum. Other equilibria can be supported by an infinitely repeated game, especially because this is also one of the situations in which cooperation may arise. But, once a Nash equilibrium is played, it must be stuck to thereafter. This is exactly what Theorem 2 needs to prove its results: the equilibrium point mentioned in Theorem 1 is a Nash equilibrium of the stage game later used by Theorem 2, from which its proof follows.

References

[1] H. Ren, B. Xu, Y. Wang, C. Yi, C. Huang, X. Kou, T. Xing, M. Yang, J. Tong, and Q. Zhang, "Time-series anomaly detection service at microsoft," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New York, NY, USA: ACM, 2019, pp. 3009–3017. [Online]. Available: http://doi.acm.org/10.1145/3292500.3330680

[2] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2018.

[3] W. Sultani, C. Chen, and M. Shah, "Real-world anomaly detection in surveillance videos," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[4] V. J. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial Intelligence Review*, vol. 22, no. 2, pp. 85–126, Oct 2004. [Online]. Available: https://doi.org/10.1007/s10462-004-4304-y

[5] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[6] T. Schlegl, P. Seeböck, S. Waldstein, U. Schmidt-Erfurth, and G. Langs, "Unsupervised anomaly detection with generative adversarial networks to guide marker discovery," 03 2017, pp. 146–157.

[7] X. Hou and L. Zhang, "Saliency detection: A spectral residual approach," in *2007 IEEE Conference on Computer Vision and Pattern Recognition*, June 2007, pp. 1–8.

[8] Z. Liu, Y. Yan, and M. Hauskrecht, "A flexible forecasting framework for hierarchical time series with seasonal patterns: A case study of web traffic," in *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, ser. SIGIR '18. New York, NY, USA: ACM, 2018, pp. 889–892. [Online]. Available: http://doi.acm.org/10.1145/3209978.3210069

[9] K. Hundman, V. Constantinou, C. Laporte, I. Colwell, and T. Söderström, "Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding," *ArXiv*, vol. abs/1802.04431, 2018.

[10] H. Wang, A. Kläser, C. Schmid, and C. Liu, "Action recognition by dense trajectories," in *CVPR 2011*, June 2011, pp. 3169–3176.

[11] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, June 2005, pp. 886–893 vol. 1.

[12] B. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision (ijcai)," vol. 81, 04 1981.

[13] J. Sivic and A. Zisserman, "Efficient visual search of videos cast as text retrieval," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 4, pp. 591–606, April 2009.

[14] J. Carreira and A. Zisserman, "Quo vadis, action recognition? a new model and the kinetics dataset," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[15] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3d convolutional networks," in *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[16] S. Shalev-Shwartz and S. Ben-David, *Uderstanding Machine Learning*. Cambridge University Press, 2014, http://www.cs.huij.ac.il/~shais/UderstandingMachineLearning.

[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[18] B. H. Juang and L. R. Rabiner, "Hidden markov models for speech recognition," *Technometrics*, vol. 33, no. 3, pp. 251–272, 1991. [Online]. Available: http://www.jstor.org/stable/1268779

[19] M. Gales and S. Young, "The application of hidden markov models in speech recognition," *Foundations and Trends in Signal Processing*, vol. 1, pp. 195–304, 01 2007.

[20] Biing-Hwang Juang, Wu Hou, and Chin-Hui Lee, "Minimum classification error rate methods for speech recognition," *IEEE Transactions on Speech and Audio Processing*, vol. 5, no. 3, pp. 257–265, May 1997.

[21] D. P. Kingma and M. Welling, "An introduction to variational autoencoders," 2019.

[22] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2672–2680. [Online]. Available: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

[23] B. Nevio and Z. Michele, *Principles of Communications Networks and Systems*. Wiley, 2011.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.

[25] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Computer Vision and Pattern Recognition (CVPR)*, 2015. [Online]. Available: http://arxiv.org/abs/1409.4842

[26] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. M. Botvinick, S. Mohamed, and A. Lerchner, "beta-vae: Learning basic visual concepts with a constrained variational framework," in *ICLR*, 2017.

[27] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 448–456. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045167

[28] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[29] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[30] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[31] "Hmmlearn," https://hmmlearn.readthedocs.io, 2010.

[32] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

[33] "Ucsd anomaly detection dataset," http://www.svcl.ucsd.edu/projects/anomaly/dataset.htm, 2008.

[34] M. Ravanbakhsh, M. Nabi, E. Sangineto, L. Marcenaro, C. S. Regazzoni, and N. Sebe, "Abnormal event detection in videos using generative adversarial nets," *2017 IEEE International Conference on Image Processing (ICIP)*, pp. 1577–1581, 2017.

[35] T. G. Dietterich, R. H. Lathrop, and T. Lozano-Pérez, "Solving the multiple instance problem with axis-parallel rectangles," *Artif. Intell.*, vol. 89, no. 1-2, pp. 31–71, Jan. 1997. [Online]. Available: http://dx.doi.org/10.1016/S0004-3702(96)00034-3

[36] B. Babenko, "Multiple instance learning: Algorithms and applications," 01 2008.

[37] J. Carreira, E. Noland, C. Hillier, and A. Zisserman, "A short note on the kinetics-700 human action dataset," *CoRR*, vol. abs/1907.06987, 2019. [Online]. Available: http://arxiv.org/abs/1907.06987

[38] S. Tadelis, *Game Theory, an introduction*. Princeton University Press, 2013.

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

Acknowledgments

Il mio più grande ringraziamento va alla mia famiglia, che mi ha sempre supportato ed incoraggiato in tutte le mie scelte più importanti, come per esempio su quale percorso universitario intraprendere.

Un altro grande grazie va a Riccardo, che sebbene nell'ultimo periodo a modo suo, è sempre presente; sto ancora aspettando il mio turno per lanciarti uova però! Un altro grazie va a tutti gli amici, molti dei quali ho avuto la fortuna di conoscere durante questo percorso universitario, reso senz'altro più leggero dal buon tempo passato assieme.

Alcuni ringraziamenti vanno anche al Professore Stefano Ghidoni, e a tutti i colleghi di Data Reply, tra i quali in particolare Michele Giusto e Daniel Manrique, per l'opportunità offerta e il supporto fornito durante lo svolgimento di questa tesi.

Il mio ultimo grazie va infine ai miei 7 coinquilini di Milano, che sono di sicuro stufi di sentire i miei discorsi sulla tesi, ma non di meno mormorano assensi, aspettando certamente una giusta vendetta in stile post laurea all'UniPD.