Travail de Fin d'Etudes : The use of learning algorithms for modeling of transport phenomena

Auteur : Estrada Peñas, Joan
Promoteur(s) : Cools, Mario; Saadi, Ismaïl
Faculté : Faculté des Sciences appliquées
Diplôme : Cours supplémentaires destinés aux étudiants d'échange (Erasmus, ...)
Année académique : 2019-2020
URI/URL : http://hdl.handle.net/2268.2/9947

LIÈGE université

THE USE OF LEARNING ALGORITHMS FOR MODELING OF TRANSPORT PHENOMENA

University of Liège - Faculty of Applied Sciences

Graduation Studies conducted for obtaining the Master's degree in Civil Engineering

Promoter: Mario Cools
Co-promoter: Ismaïl Saadi

Joan Estrada Peñas

2019/2020

Abstract

From the 1990s to the present day, transportation modeling has experienced great development thanks to numerous studies that have tried in one way or another to predict traffic flows, synthesize populations, simulate transportation demand, etc. Within the transport models, the activity-based one is the most popular nowadays, due to the great flexibility and high level of detail it provides. At the same time, in the last ten years, another field dedicated to data processing has had a great development, machine learning. Machine learning includes a wide range of algorithms and statistical models that computer systems use to perform specific tasks without using explicit instructions, relying on patterns and inference instead. It is considered as a subset of artificial intelligence. Neural Networks are the most common machine learning algorithms. Neural Networks are optimization models calibrated on the basis of sample data, known as "training data", in order to make predictions or decisions without being explicitly programmed. This project aims to bring together the two worlds. First, a review of the state of the art in transportation models is presented, comparing trip-based and activity-based models. On the other hand, a review of the state of the art of Neural Networks is also made, presenting the current most efficient and developed models. To continue, a theoretical explanation of two Neural Network-based chosen models is made, the first one consisting of a Variational Autoencoder (VAE) and the second consisting of an Autoencoder based on Long Short-Term Memory (LSTM) cells. Finally, both models are applied to a dataset stemming from the 2010 Belgian Household Daily Travel Survey (BELDAM) in order to calibrate the frameworks. The model consisting in a Variational Autoencoder will be used to generate full daily activity sequences. The model based on LSTM cells will be used to predict an individuals' next steps in an activity sequence, knowing the activities he/she has done before. The VAE achieves a very good performance both in the training phase and in the inference phase. Results show very good metrics compared to the original population, and it is also able to outperform a simpler model based on a Frequency Analysis of the dataset. On the other hand, the model based in LSTM cells it is able to train correctly with considerably good results, but when new predictions are done, results are not very accurate in some cases.

Keywords: activity sequences, daily activity-travel patterns, Variational Autoencoder, Long Short-Term Memory

Content

List of figures

List of tables

1 Introduction

Transportation englobes a set processes with the aim of travel and communicate. To be able to execute these processes, several transport modes have been developed over the years (cars, trucks, planes, bicycles, trains, etc.) which circulate through different means (roads, railways, water, air, etc.).

Transport engineering takes care of planning, designing, operating and administrating the transport infrastructures, whatever the transport mode, with the aim of delivering a safe, convenient, economic and environmentally friendly movement of goods and persons.

With the continuous growing of cities, the more frequent tendency to commute between job and home, the increased offer in out-of-home activities, etc. the need of managing transport systems has increased substantially. This has encouraged the appearance of numerous research groups devoted to study transport systems.

From the 1950's to nowadays, more and more systems and models had been done with the aim of understanding, controlling, measuring, monitoring, and forecasting traffic and transport behaviors. In the early 1990's, there was a huge expansion when several pilot projects were created in the USA (Rasouli & Timmermans, 2014).

In parallel, there is another field of science that has gained a huge development in the recent years, machine learning (ML). ML includes a wide range of algorithms and statistical models that computer systems use to perform specific tasks without explicit instructions, relying on patterns and inference instead. It is considered as a subset of artificial intelligence. Machine learning algorithms are optimization models calibrated on the basis of sample data, known as "training data", in order to make predictions or decisions without being explicitly programmed. The most used approach in ML is what is called Graph Neural Networks (GNN). GNN are computer architectures formed by layers of nodes, which act as neurons, connected between them by edges. Nodes contain information and parameters and vertices express relationships between nodes and describe their interactions.

The aim of this project is to set up two GNN models based on an activity-based transport dataset. The first model is called a Variational Autoencoder and the second one consists of an Encoder-Decoder structure based on Long Short-Term Memory cells. The first model will be used to

generate activity sequences and the second one will be used to predict the next activities an individual will do, based on the previous activities he/she has done.

At first, a comprehensive literature review is conducted in order to identify the way transport networks improvement has been operated over the past decades as well as the state-of-the-art deep learning-based algorithms. Then, the mathematical structure of a Neural Network will be defined according to the model specifications, in order to understand the underlying mechanisms and the way a GNN can help us to understand the level of service of transport networks. After describing and discussing the results, recommendations will be formulated in order to improve the performance of the models and provide a better level of service of transport networks in order to shift to more sustainable cities.

At a personal level, one of the first objectives that I wanted to pursuit when I took this project, was to introduce myself in the world of transport modeling. At first, the original idea was to make a transport study linked to DNA synthetizing algorithms. But then, thanks to a proposal of the promoters of this project, it shifted to the area of machine learning and Neural Networks. So, in addition of learning about transport, a second objective was introduced, which was learning about the functioning, state of the art and development of Neural Networks, and see how they can be applied to transport models.

2 Transport systems and models

When it comes to transport planning, it is often referred as a project which studies current and future mobility demands of goods and persons. These projects are usually preceded by movement studies and surveys and they necessarily involve the different transport modes. Planning is the fundamental part of the developing process and the organization of transport, because it is the part that allows to acknowledge the problems, design and create solutions and, finally, optimize and organize the resources in order to meet the mobility demand. In transport planning, there is not a single objective, but many. The purpose is to obtain a transport system which is efficient, safe, accessible to everybody and in accordance to the environment. It is also desirable that the transport system is consonance with the urban development. However, the transport planner faces a trade-off between the service quality and the resources that need to be used, searching for an equilibrium between the objectives achieved against the resources spent.

In order to have a good transport system, modeling of the transport is necessary. Transport modeling (or transport demand modeling) allows us to estimate the passenger's flows or vehicles in a specific transport network in each of the considered transport modes for future scenarios. There are two main modeling groups: trip-based models and activity-based models. In the first ones, the analysis unit is a trip between an origin and a destination. The second ones study the trip chains in a specific period of time (usually a day) derived from taking part in different activities along the time frame.

Transport models are always embedded in a system of other models which are integrated between them. Mainly, transport models predict the demand per mode and the network models predict how demand will affect the performance of transport provision. Also, nowadays technologies with access to information of dynamical data, big data, etc., are making possible the development of new algorithms that allow to improve substantially the predictability and adjustment of those same algorithms in the real scenario that they are modeling.

Models allow to represent processes or more complex phenomena in a simple way. In other words, they simplify reality. Transport demand modeling looks to forecast several factors for future situations. Among these factors, we can count: the number of trips made in a specific area, how trips are distributed among different areas, in which transport modes are trips done, volume of travelers in public transport networks, vehicle flows in roads, etc. To be able to carry

out these forecasts, the application of several mathematical algorithms is needed. Mathematical expressions are inferred from models that correlate variables or probabilistic models. These last ones are used because it is very complicated to find established and defined relationships to represent situations where person's decisions take place.

As a result, transport models are used in the definition of transport policies and for its planification and engineering: calculate infrastructure's capacity, estimate the financial and social viability of a project, use of cost-benefit analysis and social impact analysis, calculate environmental impacts, etc.

A transport model, must comply with several basic conditions (Friedrich, M., 2007).

First of all, it has to be executable. Depending on the phenomena that we want to model, of the results that we want to obtain and their precision and accuracy, we have to select all the relevant variables that allow to recreate the current situation in a realistic way. Among all these variables, there are some which are indispensable and there are a lot of them that, even though they may have some effect, their value is minimum or marginal and if they were considered, they would complicate in excess the model processing.

Secondly, it has to be logical and consistent. The model has to contain logical processes. The results must be coherent, measurable and they must avoid discrepancies. For example, an increase in population of an analyzed area should produce an increase in trips generation in this area.

In third place, the model has to be transparent. The results outputted by the model should be justified with expressions and mathematical terms that can be understood and controlled. A non-transparent model means that the obtained results are difficult to justify and that there is some uncertainty in the model parameters.

And finally, the model has to be sensitive to changes. This means that changes in the inputs must deliver changes in the outputs.

As mentioned before, transport modeling comes from mathematical, physical and economical principles that allow us to replicate in a rational way the behaviors of transport systems. Two theories that support this are the utility theory and the gravitation theory.

Based on the principle that individuals act rationally, in transport, a traveler will only perform a trip, if the utility of the same is higher than the one of not performing it (Fricker, Jon D. &

Whitford, Robert K., 2003). That is valid always while travelling is not a cost-free activity. Meanwhile the activity that is to be done in the destination produces some benefit to the traveler, the transport activity only generates costs. For that reason, transport engineering says that transport demand is a derived demand. When it is said that traveling is not a cost-free activity, it is not only referred at the direct costs of the trip (fuel, tolls, etc.), but also at the consumption of a very valuable resource: time. It is very common that in transport models time is considered as the main cost that users pay. Time evaluation by users is also quite complex. Previous research (Abrantes & Wardman, 2011; Athira et al., 2016; Litman, 2002) indicated that time is valued in different ways depending on for what it could be used (opportunity cost) or on the income of the person in that time (the more income, the more valued the time).

Physical laws have also proven useful to recreate the situations that occur in the transport phenomena. The gravitation law says that "the force that makes an object with mass m_1 to another object with mass m_2 is directly correlated with the product of both masses, and inversely correlated with the squared distance that separates them". In an analogic way, for trips distribution in a transport model, the quantity of trips between an origin and a destination it is directly correlated with the attraction of each destination area and inversely correlated with the generalized cost of travel between the two zones.

So as it has been mentioned before, there are two main transport models: models based in trips between an origin and a destination and models based on activities chains during a time frame. Both approaches have different ways of operating and different uses and will be explained briefly below.

2.1 Trip-based models

2.1.1 Definition

In this model (Castiglione, Joe & Gliebe, John, 2015; McNALLY, 2000), the main analysis unit is the trip *per se.* They are also known as four-steps models and they are accepted as valid tools for transport planning. This procedure, originally developed in the 1950s uses aggregated data from different subdivisions of territory to estimate trips with the current network.

The first step of the four is "trip generation". In this step, the aim is to transform demographic, economic and household attributes in each zone, in trips generated by this same zone. The total prediction of trips is made by zone. It is usual that the transport authority has some traffic analysis areas defined. Each area or zone produces or attracts trips. The number of generated

trips is associated to the population quantity and its features within the studied area, while trips attraction is related with the rest of the economic activities that are performed within the area.

The second step is the “trip distribution”. In this step, pairs are created for each one of the produced trips in the different areas in the previous step, with some of the different attraction locations in other areas or in the same one. The result is a table of trips between the different areas of the model, known as the origin-destination matrix (O-D matrix). The matrix shows the number of trips from each one of the origins “i” to each one of the destinations “j”.

The third part of the four-step model is the “mode choice”. This step divides the total number of trips between each pair of areas by transport mode. For this, detailed information of the network needs to be possessed in addition of the public and private transport provision, along with the number of trips between areas obtained in the second step. The result of this step are several matrixes with trips, one for each mode.

And finally, the last step is “route assignment”. It assigns to a route a trip between an origin and a destination in a particular transport mode. Often, Wardrop’s principle of user equilibrium is applied (equivalent to a Nash equilibrium), where each driver chooses the shortest path, subject to every other driver doing the same. The problem comes because travel times are a function of demand, while demand is a function on travel time.

After the model is set up, it is evaluated according to some decision criteria and parameters. One common criterion is cost-benefit analysis.

2.1.2 Deficiencies of trip-based models

Trip-based models have two major points that make them weak in terms of transport planning and forecasting (Castiglione, Joe & Gliebe, John, 2015).

First of all, the independence assumptions that these models made. Transportation policies and investment questions have become more complex. The ones responsible for taking decisions have to face questions not only about how and where to expand transportation system capacity, but they must also consider questions about how to best manage the existing transportation system. Trip-based models are not able to provide information to address these policy questions because they assume that all trips are made independently. Also, they lack details on individual travelers and their coordination with other household members. Because trip-based models rely on aggregation of persons and household, they are limited on their ability to represent how

different people may respond to small changes in inputs. They are limited in sensitivity (Castiglione, Joe & Gliebe, John, 2015).

And the second weakness is the aggregation bias, which refers to the assumption that group characteristics are shared by all the individuals who are members of that group. In other words, that all households of the same type behave similarly. However, there is tremendous diversity in how different types of persons and households make travel decisions depending on factors such as income, transit accessibility, competition with other household members for vehicles, travel times, etc. The use of aggregate values distorts a model's sensitivity, as mentioned before (Castiglione, Joe & Gliebe, John, 2015).

Although it may be theoretically possible to incorporate additional detail in trip-based models through the use of additional market segmentation, zones or time period, it is practically challenging because the aggregate trip-based model's reliance on two dimensional origin-destination (O-D) matrices causes model run times, storage, and memory requirements to increase exponentially as segmentation increases (Castiglione, Joe & Gliebe, John, 2015).

2.2 Activity-based models

On the other hand, activity-based models (Castiglione, Joe & Gliebe, John, 2015) have gained popularity in the last years, and they are the kind of models that will be used in this project. They share some similarities with the classical four-step model mentioned before: activities are generated, destinations for the activities are identified, travel modes are determined, and the specific network facilities or routes used for each trip are predicted.

These models are based in the idea that travel demand derives from people's needs and desires to participate in activities. In some cases, these activities are located inside their homes, but in many cases, they occur outside their homes, resulting in the need to travel. They are distinguished from trip-based models by a number of features. They represent each person's activity and travel choices across a time frame (usually a day), setting priorities among them when it comes to scheduling. As any individuals schedule becomes filled, the time available to participate in and travel do additional activities diminishes.

2.2.1 Main features of activity-based models

2.2.1.1 Individual travelers

By functioning at the level of the individual traveler, activity-based models are able to represent greater variation across the population than aggregate trip-based models. A key advantage is

that they can incorporate new explanatory variables and new sensitivities much more easily because they are typically implemented using a microsimulation framework (Castiglione, Joe & Gliebe, John, 2015).

2.2.1.2 Interrelated decision making

Activity-based models represent the interrelated aspect of activity and travel choices for all travel conducted by a person or households during a day, including purpose, location, timing, and travel modes, which results in a more detailed representation of how travelers may respond to investment and policy alternatives, as well as land use and socioeconomic changes. These models explicitly represent how individuals move from one geographic location to another during the day – the destination of one trip becomes the origin of the following trip. This geographic consistency realistically bounds where, how, and when travelers can travel. Consistency in the representation of time of day also distinguishes activity-based models from trip-based models. They also use information about tours and trips to impose plausible constraints on the travel modes that are available to travelers. For example, it is highly unlikely that a traveler who has used public transport to get to work is going to drive home alone, because he or she does not have a vehicle to use (Castiglione, Joe & Gliebe, John, 2015).

2.2.1.3 Detailed information

Activity-based models incorporate significantly more detailed input information and produce significantly more detailed outputs than trip-based models. They can use a wider range of important explanatory variables to predict travel patterns than trip-based models. For example, consistent representation of trips made jointly by household members is only possible using activity-based models. These models also include explicit and detailed models of time-of-day choices. The temporal information is especially critical given the travel demand and transportation system management policy and investment choices faced by decision makers (Castiglione, Joe & Gliebe, John, 2015).

2.2.1.4 Integrated Travel Demand Model System

Activity-based models are always embedded within an integrated model system in which there is an interaction between the activity-based or trip-based models, which predict the demand for travel, and network models, which predict how this demand affects the performance of the transportation network supply. Most activity-based models are embedded within a basic integrated model system that incorporates a limited number of essential components (Castiglione, Joe & Gliebe, John, 2015):

- Population synthesis models create detailed, synthetic representations of populations of individuals within households (agents) whose choices are simulated in activity-based models.
- Activity-based travel demand models predict the long-term choices (such as work location and automobile ownership) and the daily activity patterns of a given synthetic population, including activity purposes, locations, timing, and modes of access.
- Auxiliary models provide information about truck and commercial travel, as well as special purpose travel such as trips to and from airports or travel made by visitors.
- Network supply models are tightly linked with activity-based demand models. The flows of travel by time of day and mode predicted by activity-based travel demand models and auxiliary models are assigned to roadway, transit, and other networks to generate estimates of volumes and travel times.

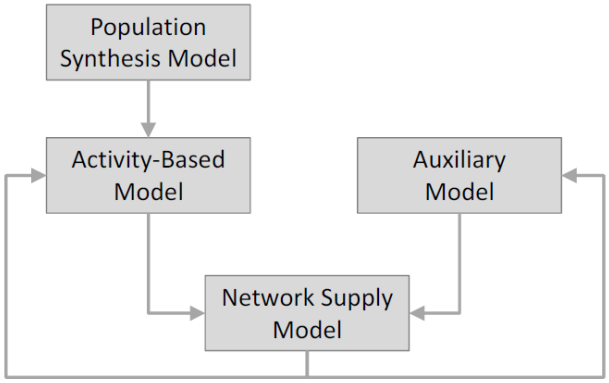

Figure 1: Basic integrated model components. (Source: Castiglione, Joe & Gliebe, John, 2015)

2.2.2 Examples of previous studies

Activity-based models developed at the beginning or currently being used today use one or more decision-making criterion, which result in a great diversity of methodologies to model travel behavior. Generally, this models include microeconomic optimizers, which assume that people tend to minimize travel time or costs, computational process models, which try to mimic human behavioral processes, naïve data-driven models, which derive models from observed data, satisficing models, which try to set the number of options available to simulated individuals, and cellular automata, which apply principles of physics to reproduce human behavior (Henson et al., 2009).

The first models that began incorporating behavioral features were developed in the late 1970's and beginning of the 1980's, such as BSP (Huigen, 1986) and the Computational Algorithms for

Rescheduling Lists of Activities (CARLA) (Jones et al., 1983). SCHEDULER (Gärling et al., 1994) was the first model to incorporate a computational process model (CPM), adding psychometric cognitive basis. In SCHEDULER, activities are selected from the long-term calendar which searches to simulate a person's long-term memory. These activities then conform a schedule that is "mentally executed".

However, it wasn't until the mid-1990's when the activity-based models started to grow in the number of models and in the number of paradigms used in them. Ettema et al. (1996) presented the Simulation Model of Activity Scheduling Heuristics (SMASH) and COMRADE (1995). "SMASH is a CPM and econometric utility-based hybrid model that focuses on the pre-trip planning process" (Henson et al., 2009). It was focused on the activity sequences, but not in its duration. On the other hand, COMRADE was more focused on this area, and it is able to add the continuous nature of decision making. The Model of Action Space in Time Intervals and Clusters (MASTIC) (Dijst & Vidakovic, 1997) identifies clusters in the action space to carry out and schedule activities. An optimization model called Household Activity Pattern Problem (HAPP) (Recker, 1995) developed a variant of the pick-up and delivery time window problem, creating activity schedules in an optimal way. Also, the GIS-Interfaced Computational-process modeling for Activity Scheduling (GISICAS) (Kwan, 1997), a simplified version of SCHEDULER, uses a Geographic Information System (GIS) to add spatial information in the model to create individual schedules, starting with high priority activities (Henson et al., 2009).

In 1998, MatSIM, an econometric utility-based model with microsimulation, was published (Balmer et al., 2008). The model uses a basic evolutionary (relaxation) strategy to develop travel patterns. Initial activity schedules are used to generate traffic inside a microsimulation. The actual travel times for the schedules are calculated and are used to update the previous schedules. The process is repeated until the final equilibrium is obtained (Henson et al., 2009). In the same year, ALBATROSS was released by Arentze and Timmermans (2000). "ALBATROSS is a multi-agent CPM that predicts the time, location, duration, and with whom activities occur as well as the type of mode utilized, subject to spatio-temporal, institutional, and household constraints" (Henson et al., 2009). It is one of the most complete activity-based models, and it includes a large number of choice features, using an accurate classification of activities while including a large series of constraints. All of these models, among others that are not mentioned here for the sake of brevity, have been used for many purposes at the time of analyzing transport phenomena.

Among the studies that have tried to determine attitudes and behaviors of people in relation with transport phenomena, Ruiz and Timmermans (2006, 2008) looked to determine how the timing and the duration of activities was affected when resolving scheduling conflicts. Mars & Ruiz also investigated (2018) which determinants influence the elimination of activities when it comes to scheduling and which determinants affect rescheduling travel mode choice. In a similar line, Ferrer and Ruiz (2014) inquired which factors influence the travel scheduling of driving trips of habitual car users.

Regarding the issue of congestion pricing, for example, it can encompass a wide variety of different pricing and toll schemes with the purpose of managing demand, improve travel time reliability, reduce congestion, and increase usage of alternative modes such as public transport or car-sharing. Activity-based models provide more flexibility to represent different alternatives, because they show more accurately how people plan and organize their days (Henson et al., 2009). The more precision we have with a model, the more important it is to evaluate how policy changes will affect transportation. In this line, Cools et al. (2011) studied the effect of road pricing on people's tendency to adapt their current travel behavior using a two-stage hierarchical model.

Another fact that can influence traveling times, modes and decisions is weather. In this line, Cools and Creemers (2013) studied the dual role of weather forecasts on changes in activity-travel behavior in the Flemish population. They found out that the forecasted weather has a significant effect, creating changes in activity-travel behavior depending on the weather forecasted. On the other hand, different methods of obtaining weather information (media source, exposure, or perceived reliability) do not impact the probability of behavioral adaptations. In a different paper (2015) they investigated the meteorological variation in revealed preference travel data, trying to investigate the impact of weather conditions on daily activity participation (trip motives) and daily modal choices in the Netherlands. In a quite pioneering study, Saadi et al. (2018) investigated the impact of river floods on travel demand based on an agent-based modeling approach in the city of Liège, Belgium. The findings showed how travel times changed in response to the variations of levels of service in the transport network and how traffic flows are re-distributed more uniformly across the network. Roads with important traffic volumes are subjected to a decrease of activity on the contrary of roads with low traffic volumes.

On a different approach, Saadi et al. (2016) used a Hidden Markov Model (HMM) to generate a synthetic population. Synthetic populations consist of a set of agents characterized by demographic and socio-economic features which are used in micro-simulation travel demand and land use models. This HMM model outperformed the main groups of population synthesis techniques, which are fitting methods and combinatorial optimization methods, capturing the complete heterogeneity of the micro-data contrary to the standard fitting approaches. Following this study, they also developed an integrated framework for forecasting travel behavior using Markov Chain Monte Carlo simulation and profile HMMs, effectively capturing the behavioral heterogeneity of travelers (Saadi, Mustafa, Teller, & Cools, 2016).

Henson et al. (2009) reviewed 53 activity-based models as candidates for operational studies, disaster preparedness, and homeland security applications. They concluded that there has been a great progress in the last 35 years in travel demand modeling and simulation. However, models are being concentrated at two poles. They are either designed for the very short term such a daily activity pattern or for the long term with yearly cycles. They proposed that with proper data collection and analysis, other temporal dimensions such as weekly, monthly and seasonal regularities could be also dealt with.

More recently, Anda et al. (2017) studied how transport modelling should be in the age of Big Data. Since new data sources are available, such as mobile phone call records, smart card data in public transport systems and geo-coded social media records, we are able to understand mobility behavior on an unprecedented level of detail. However, despite the availability of all these Big Data sources, transport models still, most of them, are based on conventional data such as travel diaries and population census, and they normally only represent a small sample of the population (around 1%) and they are usually only updated every 5-10 years. In the study, they found out that future research should focus also in machine learning approaches which are able to mine data from complex datasets, such as the ones mentioned before, to integrate them with agent-based simulations.

So, in summary, we can see how different studies tried to address different topics related to transport phenomena in order to obtain better information for traffic planners and decision makers. However, there is still a lot of research than can be done, and for sure that machine learning techniques operating with neural networks can be of great utility to improve the existing models or even to create new ones.

3 Graph neural networks

Graph Neural Networks (GNNs) are a kind of mathematical representation that has gained increased popularity in the recent years. The areas where they can be applied are very varied, and they don't seem to stop growing: social networks, knowledge graphs, sequence prediction, text managing, image recognition, etc.

The recent success of neural networks has boosted research on data mining and pattern recognition. Many machine learning tasks such as object detection (Redmon et al., 2016), machine translation (Luong et al., 2015; Yonghui Wu et al., 2016), and speech recognition (Hinton et al., 2012), which once heavily relied on handcrafted feature engineering to extract informative feature sets, have recently been revolutionized by various end-to-end deep learning paradigms, e.g., convolutional neural networks (CNNs) (Lecun & Bengio, 1995), recurrent neural networks (RNNs) (Hochreiter & Schmidhuber, 1997), and autoencoders (Vincent et al., 2010).

The success of deep learning in many domains is in part thanks to the fast development there has been in computational resources (e.g., GPU), the availability of big training data (Anda et al., 2017), and the effectiveness of deep learning to extract latent representations from Euclidean data (e.g., images, text, and videos).

While deep learning effectively captures hidden patterns of Euclidean data, there is an increasing number of applications where data are represented in the form of graphs. For examples, in e-commerce, graph-based learning system can exploit the interactions between users and products to make highly accurate recommendations. In chemistry, molecules are modeled as graphs, and their bioactivity needs to be identified for drug discovery. In a citation network, papers are linked to each other via citationships and they need to be categorized into different groups. The complexity of graph data has imposed significant challenges on existing machine learning algorithms (Z. Wu et al., 2019).

3.1 Background

As mentioned in the research done on graph neural networks (Z. Wu et al., 2019), back in the 1990s, some studies (Sperduti & Starita, 1997) firstly applied neural networks do directed acyclic graphs, and this started to motivate early studies on GNNs. The notion of graph neural networks was initially outlined in Gori et al.(2005) and further elaborated in Scarselli et al. (2009), and Gallicchio & Micheli (2010). These early studies fall into the category of recurrent graph neural

networks (RecGNNs). They learn a target node's representation by propagating neighbor information in an iterative manner until a stable point is reached.

Thanks to the success of convolutional neural networks (CNNs) in the field of computer vision, a high number of methods that re-define the idea of convolution for data set in graphs are parallelly developed. These insights are imbricated in the convolutional graph neural networks (ConvGNNs). This kind of networks are separated into two principal groups, the spectral-based and the spatial-based approaches.

First important study on spectral-based ConvGNNs was presented in the article "Spectral Networks and locally connected networks on graphs" (Bruna et al., 2014). This article developed a graph convolution based on the theory of spectral graphs. Since that moment, there have been great improvements, approximations, end extensions on spectral-based ConvGNNs (Defferrard et al., 2017; Henaff et al., 2015; Kipf & Welling, 2017; Levie et al., 2018).

On the other hand, investigation on spatial-based ConvGNNs started much earlier than spectral-based ConvGNNs. In 2009 a study (Alessio Micheli, 2009) first addressed graph mutual dependency by architecturally composite non-recursive layers while adopting ideas of message passing from RecGNNs. Until nowadays, many spatial-based ConvGNNs have appeared (Atwood & Towsley, 2016; Gilmer et al., 2017; Niepert et al., 2016).

Apart from the mentioned RecGNNs and ConvGNNs, many alternative GNNs have been developed in the recent years. For example, we can find graph autoencoders (GAEs) and spatial-temporal graph neural networks (STGNNs). These two learning structures can be built on RecGNNs, ConvGNNs, or any other structure for graph modeling.

3.2 Definitions

Based on the study done in the paper by (Z. Wu et al., 2019), some mathematical and descriptive definitions will be done to better understand how GNNs work.

3.2.1 Graph

A graph is represented as $G = (V, E)$ where V is the set of vertices or nodes (we will refer mainly as nodes in this study), and E is the set of edges. Let $v_i \in V$ to denote a node and $e_{ij} = (v_i, v_j) \in E$ to denote an edge pointing from v_j to v_i. The neighborhood of a node v is defined as $N(v) = \{u \in V | (v, u) \in E\}$. The adjacency matrix A is a $n\ x\ n$ matrix with $A_{ij} = 1$ if $e_{ij} \in E$ and $A_{ij} = 0$ if $e_{ij} \notin E$. Nodes in the graph can have attributes X, where $X \in R^{n\ x\ d}$ is a node

feature matrix with $x_v \in R^d$ representing the feature vector of a node v. Meanwhile, a graph may have edge attributes X^e, where $X^e \in R^{m\,x\,c}$ is an edge feature matrix with $x^e_{v,u} \in R^c$ representing the feature vector of an edge (v, u).

3.2.2 Directed Graph

If a graph is directed, it means that all edges are directed from one to another. Otherwise, if there is a pair of edges with inverse directions, it may be considered as a special case of directed graphs. In that case they are undirected graphs. A graph is directed if and only if the adjacency matrix is symmetric.

3.2.3 Spatial-Temporal Graph

This kind of graph is an attributed graph where the attributes that the nodes have change dynamically over time. It is defined as $G^{(t)} = (V, E, X^{(t)})$ with $X^{(t)} \in R^{n\,x\,d}$.

3.2.4 Neurons

Each node of the graph acts as a neuron, inspired in the behavior observed in the axons of neurons in the brains. A neuron is a local computing gadget that gets an input vector $x_i \in \mathbb{R}^n$, combines it with the local parameters (weights, $w_i \in \mathbb{R}^n$, and biases, $b \in \mathbb{R}^n$) and outputs a scalar number as a result. The input vector is a combination from the previous n neurons. Once the neuron state, $z = \sum_{i=1}^{n} x_i\ w_i + b$, has been calculated, it is passed through the activation function to get the output. The output can be the final result or can serve as an input to another connected neuron.

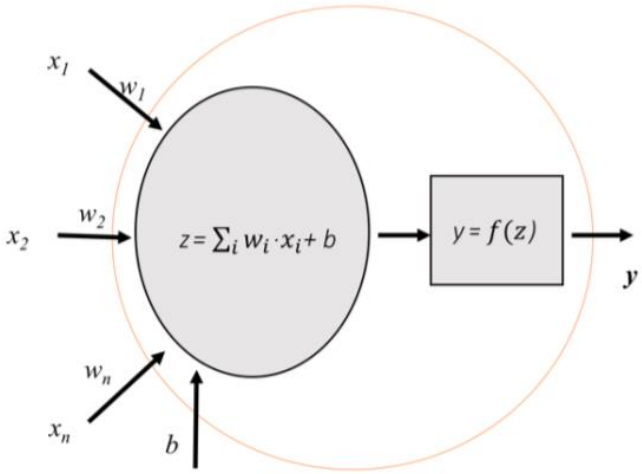

Figure 2: Schema of a neuron functioning (Source: (Torres, 2012))

3.2.5 Activation functions

Activation functions are used to scale the neuron state to decide if the neuron has to output a value or not, and in which measure. The most common activation functions used in the machine learning field are: sigmoid, SoftMax, hyperbolic tangent (tanh), and Rectified Linear Unit (ReLU).

The sigmoid (also known as "logistic") scales the values between 0 and 1. It is very used in models where the output prediction is a probability, since probability is a value that goes from 0 to 1.

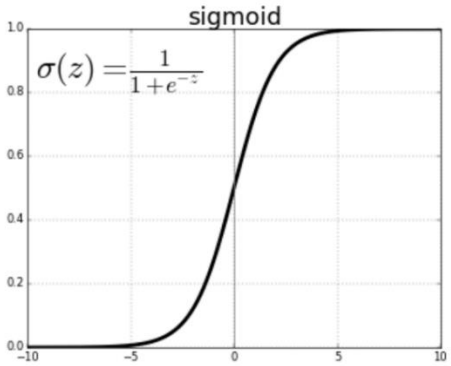

Figure 3: Sigmoid function. Very negative values will be clamped to zero, while very positive numbers will pass through.

The SoftMax function is a more generalized sigmoidal function that is used in multiclass classification problems (Sharma, 2019).

The tanh activation function is similar than the sigmoid function but the output values are ranged from -1 to 1. The advantage is that very negative values will be mapped negatively and the zero inputs will be mapped close to zero. It is very used in classification problems between two classes of data.

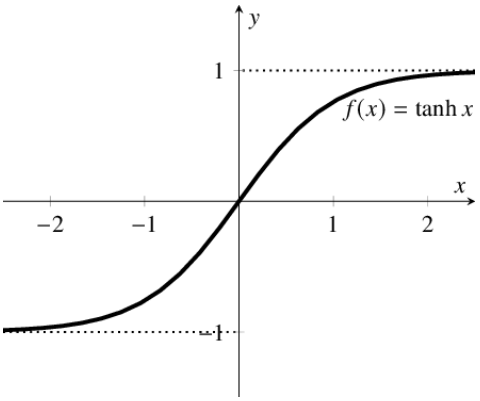

Figure 4: Hyperbolic tangent function (tanh). Very negative values will be clamped to -1, while very positive numbers will pass as a 1

Finally, the ReLU is the most used activation function in the machine learning area currently. It is used in most of the convolutional neural networks. The ReLU is half rectified. $f(z)$ is zero when z is negative, and $f(z) = z$ when z is positive. The biggest advantage of ReLU is that the gradient is not-saturated, which helps to accelerate convergence in stochastic gradient descent compared to sigmoid and tanh function (Krizhevsky et al., 2017).

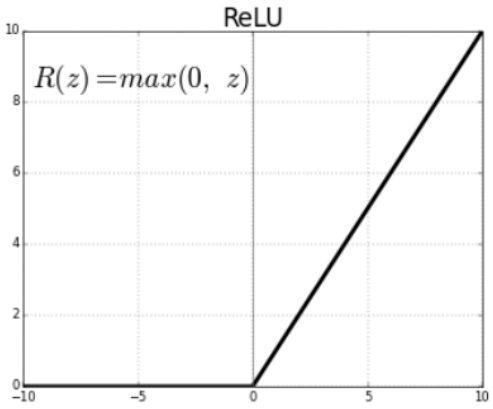

Figure 5: Rectified Linear Unit function (ReLU). Very negative values are clamped to 0, while positive values remain the same

3.2.6 Layers

Neural networks (NN) are composed of different layers of neurons. Normally, the neurons or nodes of one layer are only connected to the neurons of the next layer. So, there are three kinds of layers:

- **Input layer**: It is the first layer of a NN representing the input of the model. Usually it is not counted as a layer since it doesn't do any operation, it just reads in the data that the model is going to process.
- **Output layer**: It is the last layer of a NN, which computes the final result of the model using a neuron for each output variable.
- **Hidden layers**: They are the layers in between the input and the output, and they can be of all sorts and perform different kind of operations.

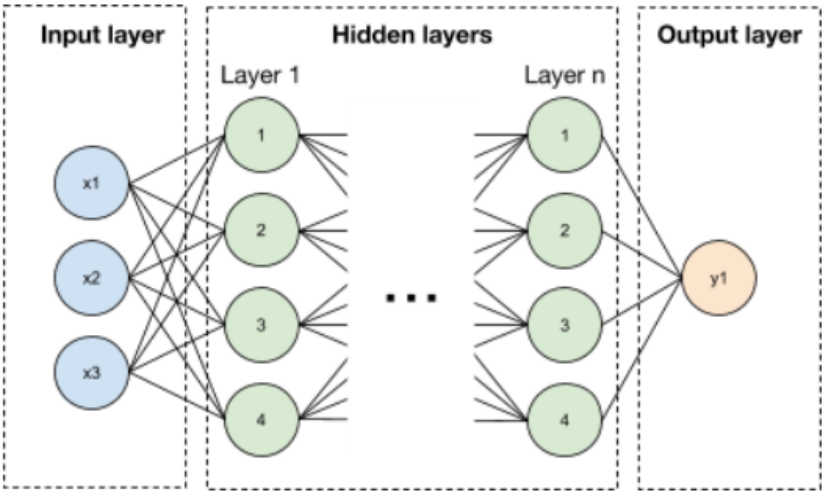

Figure 6: Schematic of a neural network with the different layers (Source: (Blaauw & Emerencia, 2016))

3.2.7 Back Propagation

The back-propagation algorithm is based on generalizing the Widrow-Hoff learning rule. It works with supervised learning, that is that the algorithm is provided with the inputs and the outputs that the network should complete, and after that, the error is calculated. The back-propagation algorithm starts with random weights and biases, and the objective is to tune them in order to reduce this error until the network has learned the training data. The standard back propagation is a gradient descent algorithm where the weights of the network are moved along the negative of the gradient performance function, in order to reduce it. The weights are changed each time the whole network is calculated. After tuning them, the best combination of weights that minimizes the error function is considered a solution to the learning problem (Puig-Arnavat & Bruno, 2015).

3.3 Categorization

According to the paper by (Z. Wu et al., 2019), graph neural networks can be classified into four big groups, which have already been mentioned before: recurrent graph neural networks (RecGNNs), convolutional graph neural networks (ConvGNNs), graph autoencoders (GAEs), and spatial-temporal graph neural networks (STGNNs). We will describe each one of them shortly.

3.3.1 Recurrent graph neural networks (RecGNNs)

They are the main pioneer works of graph neural networks. This kind of graphs assume that a node is permanently exchanging information/messages with its neighbors until a stable equilibrium is achieved. The concept of RecGNNs is pretty important in the field and it inspired

posterior research on ConvGNNs. At the beginning, due to computational limitations, the first researches focused normally on directed acyclic graphs (A. Micheli et al., 2004; Sperduti & Starita, 1997).

In RecGNNs, a node's hidden state is recurrently updated by

$$h_v^{(t)} = \sum_{u \in N(v)} f\,(x_v,\ x_{(v,u)}^e, x_u, h_u^{(t-1)})$$

where $f(\cdot)$ is a parametric function, and $h_v^{(0)}$ is initialized randomly. Thanks to the sum operation GNN are able to be applied to all nodes, even if the number of neighbors is different and the ordering of the neighborhood is not known. Then, when a convergence criterion is satisfied, the last step node hidden states are sent to a readout layer. GNN makes an alternance of the stage of node state propagation and the stage of parameter gradient computation to minimize a training objective. This strategy enables GNN to handle cyclic graphs (Z. Wu et al., 2019).

On another direction, Gated Graph Neural Network (GGNN) (Yujia Li et al., 2017) use a gated recurrent unit (GRU) (Cho et al., 2014) as a recurrent function, achieving a diminution in the recurrences to a fixed number of steps. The main advantage is that it no longer demands to constrain parameters in order to converge. A node hidden state is updated by its previous hidden states and its neighboring hidden states:

$$h_v^{(t)} = GRU(\, h_v^{(t-1)}, \sum_{u \in N(v)} W h_u^{(t-1)})$$

where $h_v^{(0)} = x_v$. GGNN uses the back-propagation through time (BPTT) algorithm to learn model parameters.

This BPTT algorithm can be a problem for large graphs. As GGNN needs to run the recurrent function multiple times over all nodes, it needs to store in the memory the intermediate states of all nodes.

To overcome this, Stochastic Steady-state Embedding (SSE) proposes a learning algorithm that is more scalable to large graphs (Dai et al., 2018). SSE updates node hidden states recurrently in a stochastic and asynchronous fashion. It alternatively samples a batch of nodes for updating the states and a batch of nodes for gradient computation (Z. Wu et al., 2019).

3.3.2 Convolutional graph neural networks (ConvGNNs)

Convolutional graph neural networks make a generalization of the operation of convolution from grid data to graph data. The key is to generate a node v's representation by aggregating its own features x_v and neighbors' features x_u where $u \in N(v)$. In contrast with RecGNNs, ConvGNNs stack multiple graph convolutional layers to extract high-level node representations.

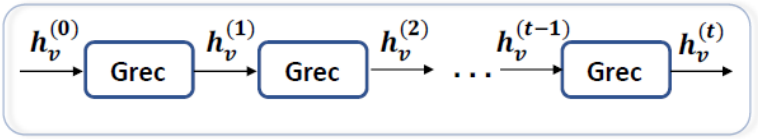

Figure 7: Recurrent graph neural networks (RecGNNs) (Source: (Z. Wu et al., 2019))

As we can observe in Figure 7, RecGNNs use the same graph recurrent layer (Grec) in updating node representations. On the other hand, as it can be seen in Figure 8, ConvGNNs use a different graph convolutional layer (Gconv) in updating node representations.

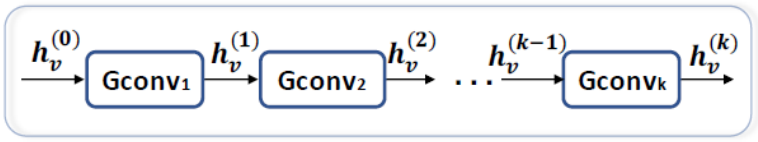

Figure 8: Convolutional Graph Neural Networks (ConvGNNs) (Source: (Z. Wu et al., 2019))

Graph convolutions are more convenient and efficient to mix with other neural networks, and for this reason the popularity of ConvGNNs has had a fast increase in recent years. There are two categories for ConvGNNs, the spectral-based and the spatial-based ones.

3.3.2.1 Spectral-based ConvGNNs

Methods based in the spectral-based principle have a firm mathematical background in graph signal processing (S. Chen et al., 2015; Sandryhaila & Moura, 2013; Shuman et al., 2013). They make the assumption of graphs to be undirected. The normalized graph Laplacian matrix is a mathematical representation of an undirected graph, defined as $L = I_n - D^{-1/2}AD^{-1/2}$, where D is a diagonal matrix of node degrees, $D_{ii} = \sum_j(A_{ij})$. This matrix L has the property of being real symmetric positive semidefinite. With this property, the normalized Laplacian matrix can be factored as $L = U\Lambda U^T$, where$U = [u_0, u_1, \dots, u_{n-1}] \in R^{n\,x\,n}$ is the matrix of eigenvectors ordered by eigenvalues and Λ is the diagonal matrix of eigenvalues (spectrum), $\Lambda_{ii} = \lambda_i$. The eigenvectors of the normalized Laplacian matrix form an orthonormal space, so mathematically speaking, $U^T U = I$. In graph signal processing, a graph signal $x \in R^n$ is a

feature vector of all the nodes of a graph where x_i is the value of the i^{th} node. The graph Fourier transform to a signal x is defined as $\mathcal{F}(x) = U^T x$, and the inverse graph Fourier transform is defined as $\mathcal{F}^{-1}(\hat{x}) = U^T \hat{x}$, where $\hat{x}$ represents the resulted signal from the graph Fourier transform. The graph Fourier transform arranges the input graph signal to the orthonormal space where the basis is formed by eigen vectors of the normalized graph Laplacian. The elements of the transformed signal $\hat{x}$ are the coordinates of the graph signal in the new space so that the input signal can be represented as $x = \sum_i \hat{x}_i u_i$, that is exactly the inverse graph Fourier transform. Then, the graph convolution of the input signal x with a filter $g \in R^n$ is defined as:

$$x *_G g = \mathcal{F}^{-1}\left(\mathcal{F}(x) \odot \mathcal{F}(g)\right) = U(U^T x \odot U^T g)$$

where $\odot$ denotes the element-wise product. If we denote a filter as $g_\theta = diag(U^T g)$, then the spectral graph convolution is simplified as:

$$x *_G g_\theta = U g_\theta U^T x$$

Spectral based ConvGNNs all follow these definitions made. The main difference lies in choosing the filter g_θ (Z. Wu et al., 2019).

This filter $g_\theta = \Theta_{i,j}^{(k)}$ is a set of learnable parameters and considers graph signals with multiple channels. The graph convolutional layer of Spectral CNN is defined as:

$$H_{:,j}^{(k)} = \sigma\left(\sum_{i=1}^{f_{k-1}} U\Theta_{i,j}^{(k)} U^T H_{:,i}^{(k-1)}\right) for\ j = 1, 2, \ldots, f_k$$

where k is the layer index, $H^{(k-1)} \in R^{n\, x\, f_{k-1}}$ is the input graph signal, $H^0 = X, f_{k-1}$ is the number of input channels and f_k is the number of output channels, $\Theta_{i,j}^{(k)}$ is a diagonal matrix filled with parameters that can be learned (Z. Wu et al., 2019).

Due to the eigen-decomposition of the Laplacian matrix, Spectral CNN faces three constraints or drawbacks. First of all, any disturbance or modification to a graph results in a change of eigen basis. Secondly, the learned filters are dependent on the domain, so they cannot be applied to a graph with a different structure. And finally, eigen-decomposition requires $O(n^3)$ computational complexity.

3.3.2.2 Spatial-based ConvGNNs

Comparable to the convolutional operation of a conventional CNN of an image, spatial-based methods define graph convolutions based on the spatial relationships of a node. Images can be taken as a special kind of graph where each pixel represents a node. Each pixel is directly connected to its nearby pixels, as it can be seen in Figures 9 and 10. The spatial-based graph convolutions convolve the central node's representation with its surrounding nodes to obtain the updated representation for the central node, as shown in Figure 9. From another point of view, spatial-based ConvGNNs participate in the same idea of information/message passing with RecGNNs. The spatial graph convolutional operation basically propagates node information along edges (Z. Wu et al., 2019).

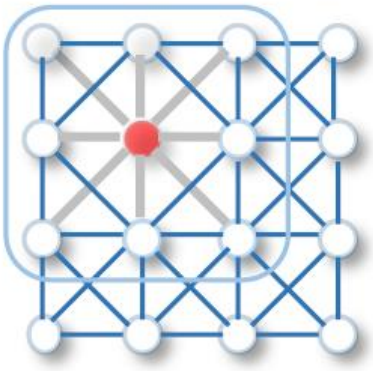

Figure 9: 2D Convolution (Source: (Z. Wu et al., 2019))

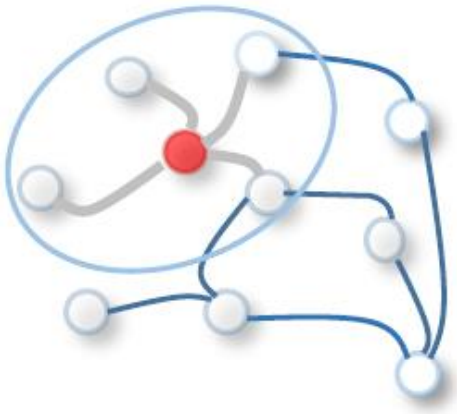

Figure 10: Graph Convolution. Different from 2D Convolutions, the neighbors of a node are unordered and variable in size (Source: (Z. Wu et al., 2019))

Neural Network for Graphs (NN4G) (Alessio Micheli, 2009), proposed in parallel with GNN, is the first work in the direction of spatial-based ConvGNNs. This kind of neural network learns graph mutual dependency through incremental building of the architecture. NN4G does graph convolutions by summing up the neighborhood of a node's information directly. The next layer node states in NN4G are generated by:

$$H^{(k)} = f(W^{(k)^T} x_v + \sum_{i=1}^{k-1} \sum_{u \in N(v)} \Theta^{(k)^T} h_u^{(k-1)})$$

where $f(\cdot)$ is an activation function and $h_v^{(0)} = 0$. This equation can also be written in a matrix form:

$$H^{(k)} = f(XW^{(k)} + \sum_{i=1}^{k-1} AH^{(k-1)}\Theta^{(k)}$$

which resembles the form of Graph Convolutional Network (Kipf & Welling, 2017).

On a slight different approach, Diffusion Convolutional Neural Network (DCNN) (Atwood & Towsley, 2016) regards graph convolutions as a diffusion process. It considers that information is sent from one node to one of its neighbors with a certain transition probability, with the aim that information allocation can reach equilibrium after several rounds. DCNN defines the diffusion graph convolution as:

$$H^{(k)} = f\left(W^{(k)} \odot P^k X\right)$$

where $f(\cdot)$ is an activation function and the probability transition matrix $P \in R^{n\,x\,n}$ is computed by $P = D^{-1}A$ (Z. Wu et al., 2019).

Diffusion Graph Convolution (DGC) (Yaguang Li et al., 2018) summarizes outputs at each diffusion step instead of concatenation. It defines the diffusion graph convolution by:

$$H = \sum_{k=0}^{K} f(P^k X W^{(k)})$$

where $W^{(k)} \in R^{D\,x\,F}$ and $f(\cdot)$ is again an activation function (Z. Wu et al., 2019).

Also, it is important to mention that with the years, there have been improvements in terms of training efficiency. Training ConvGNNs such as GCN (Atwood & Towsley, 2016) normally requires to save the whole graph data and intermediate states of all nodes into memory. The algorithm that trains the full-batch for ConvGNNs struggles considerably with the memory overflow problem, especially when a graph contains millions of nodes. To save memory, GraphSage (Hamilton et al., 2018) presents a batch-training algorithm for ConvGNNs. The algorithm samples a tree rooted at each node by recursively expanding the root node's neighborhood by K steps with a sample size that is fixed. For each sampled tree, GraphSage calculates the root node's hidden representation by aggregating hidden nodes representations from bottom to top in a hierarchical way (Z. Wu et al., 2019).

Fast Learning with Graph Convolutional Network (FastGCN) (Jie Chen et al., 2018) samples a fixed number of nodes for each graph convolutional layer instead of sampling a fixed number of neighbors for each node like GraphSage does. They use Monte Carlo approximation and variance

reduction techniques to make the training process easier. Since FastGCN samples nodes independently for each layer, connections between layers are possibly few and scattered (Z. Wu et al., 2019).

In a different project, Stochastic Training of Graph Convolutional Networks (StoGCN) (Jianfei Chen et al., 2018) makes a reduction on the size of the receptive field of a graph convolution to an arbitrary small scale using historical node representations as a control variate. Nonetheless, StoGCN still has to save intermediate states of all nodes, which for large graphs, it consumes a lot of memory (Z. Wu et al., 2019).

To continue, ClusterGCN (Chiang et al., 2019) samples a subgraph using an algorithm that clusters graphs and performs graph convolutions to nodes within the samples subgraph. Since the neighborhood search is constrained within the mentioned subgraph, ClusterGCN is capable of handling larger graphs and using deeper architectures at the same time, in less time and with less memory than the previous ones (Z. Wu et al., 2019).

3.3.2.3 Comparison between spectral-based and spatial-based models

Spectral models possess a theoretical foundation in graph signal processing. While they design new graph signal filters new ConvGNNs can be built. On the other hand, though, spatial models are preferred over spectral models thanks to their higher efficiency, generality, and flexibility issues.

Spectral models are less efficient than spatial models because they need to perform eigenvector computation or handle the whole graph at the same time. Spatial models are more suitable for large graphs because they perform convolutions in the graph domain via information propagation. Computations can be carried out in a batch of nodes instead of the whole graph, and that saves time and memory.

Secondly, spectral models which rely on a graph Fourier basis generalize poorly to new graphs, since they assume an invariable or fixed graph. Any perturbations and changes to the graph would result in a change of the eigenbasis. On the other hand, spatial-based models make graph convolutions locally on each node where weights can be shared easily across different locations and structures.

Thirdly, and finally, spectral-based models are limited to operate on undirected graphs. Spatial-based models can handle easier multi-source graph inputs such as edge inputs, directed graphs, signed graphs, and heterogeneous graphs (Z. Wu et al., 2019).

3.3.2.4 Graph Pooling Modules

After a Graph Neural Network generates node features, they can be used for the final task. However, using all these features directly can be computationally challenging, and for this reason, a down-sampling strategy is needed. Depending on what we want to achieve and the role this strategy takes in the neural network, different names are adopted.

First, the pooling operation aims to reduce the number of parameters by down-sampling the nodes in order to generate smaller representations and achieving a firm structure ready to avoid overfitting, permutation invariance, and computational complexity issues.

On the second place, the readout operation is mainly used to generate graph-level representation based on node representations (Z. Wu et al., 2019).

Nowadays, mean/max/sum pooling is the oldest and effective way to implement down-sampling because calculating the mean/max/sum value in the pooling window is fast:

$$h_G = mean/max/sum(h_1^{(K)}, h_2^{(K)}, \dots, h_n^{(K)})$$

where K is the index of the last graph convolutional layer (Z. Wu et al., 2019).

In the paper "Deep Convolutional Networks on Graph-Structured Data" (Henaff et al., 2015) it is shown that performing a simple max/mean pooling at the beginning of the network is especially important to reduce the dimensionality in the graph and reduce also the cost of the expensive Fourier transform operation. In addition, some works (Gilmer et al., 2017; Yujia Li et al., 2017) also use attention mechanisms to enhance the mean/sum pooling (Z. Wu et al., 2019).

Attention mechanisms are used when in dealing with a node, we want to focus more on a specific area, whether it is the node's surroundings or another one, because the contribution of the information present in this area is more valuable than the information other areas may have.

In general, pooling is an essential operation to reduce graph size. However, it is still an open question how to improve the effectiveness and how to deal with the computational complexity of pooling (Z. Wu et al., 2019).

3.3.3 Graph autoencoders (GAEs)

This kind of graphs are unsupervised learning structures which encode nodes/graphs into a latent vector space and then reconstruct graph data from the encoded information. GAEs are used to learn network embeddings and graph generative distributions, basically.

3.3.3.1 GAEs for Network Embedding

When we talk about network embedding, we refer to a low-dimensional vector representation of a node which preserves a node's topological information. Topology is the branch of mathematics which studies the properties of geometrical bodies which remain unaltered by continuous transformations. Also, mathematicians use topology as the reference of a certain family of subgroups of a given set, a family that follows rules about unions and intersections. GAEs learn network embeddings using an encoder to extract network connections to preserve the graph topological information such as the PPMI (Positive Pointwise Mutual Information) matrix and the adjacency matrix.

The first works on this line mainly employ multi-layer perceptrons to build GAEs for network embedding learning (Z. Wu et al., 2019). Multi-layer perceptrons refer generally as any kind of feedforward neural network, similar to the ones that we've been talking all the time. They consist of at least one input layer, a hidden layer and an output layer.

Deep Neural Network for Graph Representations (DNGR) (Cao et al., 2016) uses a stacked denoising autoencoder (Vincent et al., 2008) to encode and decode the PPMI matrix via multi-layer perceptrons (Z. Wu et al., 2019).

Structural Deep Network Embedding (SDNE) (Wang et al., 2016) makes use of a stacked autoencoder to preserve the node first-order proximity and second-order proximity together. This network suggests two loss functions on the outputs of the encoder and the outputs of the decoder separately. The loss function calculates the difference between the expected output and the given current output, in a supervised learning way. The first loss function enables the learned network embeddings to preserve the node first-order proximity by minimizing the separation between a node's network embedding and its neighbor's network embeddings. This loss function is L_{1st} is defined as:

$$L_{1st} = \sum_{(v,u)\in E} A_{v,u} \|enc(x_v) - enc(x_u)\|^2$$

where $x_v = A_{v,:}$ and $enc(\cdot)$ is an encoder which consists of a multi-layer perceptron (Z. Wu et al., 2019).

The second loss function allows the learned network embeddings to preserve the node second-order proximity by minimizing the distance between a node's inputs and its reconstructed inputs. Concretely, the second loss function L_{2nd} is defined as:

$$L_{2nd} = \sum_{v \in V} \left\| dec\big(enc(x_v)\big) - x_v) \odot b_v \right\|^2$$

where $b_{v,u} = 1$ if $A_{v,u} = 0$, $b_{v,u} = \beta > 1$ if $A_{v,u} = 1$, and $dec(\cdot)$ is a decoder which consists of a multi-layer perceptron (Z. Wu et al., 2019).

DNGR and SDNE consider only node structural information which is about the connectivity between pairs of nodes. They ignore that nodes may contain feature information that depicts the attributes of nodes themselves. Graph Autoencoder (GAE*[1]) (Kipf & Welling, 2016) uses GCN (Kipf & Welling, 2017) to encode structural information of a node and feature information of the same node at the same time. The encoder of GAE* is based in two graph convolutional layers, which take the form:

$$Z = enc(X, A) = Gconv(f\big(Gconv(A, X;\ \Theta_1)\big);\ \Theta_2)$$

where Z denotes the network embedding matrix of a graph, $f(\cdot)$ is a Rectified Linear Unit (ReLU) activation function and the $Gconv(\cdot)$ function is a graph convolutional layer. GAE* searches to decode node relational information from their embeddings by reconstructing the graph adjacency matrix, which is defined as:

$$\hat{A}_{v,u} = dec(z_v, z_u) = \sigma(z_v^T z_u)$$

where z_v is the embedding of node v. GAE* is trained by minimizing the negative cross entropy given the real adjacency matrix A and the reconstructed adjacency matrix $\hat{A}$ (Z. Wu et al., 2019).

But, only by reconstructing the graph adjacency matrix we can face problems of overfitting due to autoencoders' capacity. Variational Graph Autoencoder (VGAE) (Kipf & Welling, 2016), which will be explained in further detail later, is a variational version of GAE to learn the distribution on data. VGAE optimizes the variational lower bound L:

$$L = E_{q(Z|X,A)}[\log p(A|Z)] - KL[q(z|X, A||p(Z]$$

where $KL(\cdot)$ is the Kullback-Leibler divergence function which measures the distance between two distributions, $p(Z)$ is a Gaussian prior $p(Z) = \prod_{i=1}^{n} p(z_i) = \prod_{i=1}^{n} N(z_i|0, I)$, $p\big(A_{ij} = 1\big|z_i, z_j\big) = dec\big(z_i, z_j\big) = \sigma(z_i^T z_j)$, $q(Z|X, A) = \prod_{i=1}^{n} q(z_i|X, A)$ with $q(z_i|X, A) =$

[1] We name it GAE* to avoid ambiguity with the global definition

$N(z_i|\mu_i, diag(\sigma_i^2))$. The mean vector μ_i is the i^{th} row of an encoder's outputs and $\log \sigma_i$ is derived similarly as μ_i with another encoder. VGAE assumes the empirical distribution $q(Z|X, A)$ should be as close as possible to the prior distribution $p(Z)$.

For the methods mentioned before, they basically learn network embeddings by solving a link prediction problem. However, the sparsity of a graph originates the number of positive node pairs to be far less than the number of negative node pairs. To minimize this problem, another line of works converts a graph into sequences by random permutations or random walks. This way, machine learning approaches which are related to sequences can be used to deal with graphs. Deep Recursive Network Embedding (DRNE) (Tu et al., 2018) assumes a node's network embedding should approximate the aggregation of its neighborhood network embeddings. It uses a Long Short-Term Memory (LSTM) network (Hochreiter & Schmidhuber, 1997), which will be explained in detail later, to aggregate a node's neighbors. DRNE learns network embeddings via an LSTM network rather than using the LSTM network to generate network embeddings. It avoids the problem that LSTM's are not invariant to the permutation of node sequences (Z. Wu et al., 2019).

3.3.3.2 GAEs for Graph Generation

When we have multiple graphs, GAEs can learn the generative distribution of graphs by encoding them into hidden representations and decoding a graph structure given hidden representations. Most GAEs are designed to solve the molecular graph generation problem, which is very useful in drug discovery.

There are some approaches that generate graphs by proposing nodes and edges step by step, in a sequential way. Also, alternative solutions are applicable to general graphs by iteratively adding nodes and edges to a growing graph until a certain criterion is satisfied. Deep Generative Model of Graphs (DeepGMG) (Yujia Li et al., 2018) assumes the probability of a graph is the sum of all possible node permutations:

$$p(G) = \sum_{\pi} p(G, \pi)$$

where π denotes a node ordering. These models apprehend the complex joint probability of all nodes and edges in the graph. DeepGMG creates graphs by making a sequence of decisions, such as: whether to add a node, which node to add, whether to add an edge, and which node

to connect to the new one. The process of deciding all this things, is conditioned on the node states and the graph state of a growing graph updated by a RecGNN (Z. Wu et al., 2019).

Some other approaches operate in a more global way, they output a graph all at once instead of building it sequentially. Graph Variational Autoencoder (GraphVAE) (Simonovsky & Komodakis, 2018) models the existence of nodes and edges as independent variables which are random. The GraphVAE optimizes the variational lower bound:

$$L(\phi, \theta; G) = E_{q_\phi(z|G)}[-\log p_\theta(G|z)] + KL[q_\phi(z|G)||p(z)]$$

where $p(z)$ follows a Gaussian prior, ϕ and θ are learnable parameters. To do this, it assumes the posterior distribution $q_\phi(z|G)$ defined by an encoder and the generative distribution $p_\theta(G|z)$ defined by a decoder (Z. Wu et al., 2019).

3.3.4 Spatial-temporal graph neural networks (STGNNs)

This last kind of structure of graph neural network looks for learning dynamic hidden patterns from spatial-temporal graphs. This kind of graphs are becoming more important over the years thanks of their varied application in many fields such as traffic speed forecasting (Yaguang Li et al., 2018), driver maneuver anticipation (Jain et al., 2016), and human action recognition (B. Yu et al., 2018).

There are two main approaches in STGNNs: RNN-based methods and CNN-based methods.

Most RNN-based approaches capture spatial-temporal dependencies by filtering inputs and hidden states passed to a recurrent unit using graph convolutions (Yaguang Li et al., 2018; Seo et al., 2016; Zhang et al., 2018). To better understand this, suppose a simple RNN which takes the form:

$$H^{(t)} = \sigma(WX^{(t)} + UH^{(t-1)} + b)$$

where $X^{(t)} \in R^{n\,x\,d}$ is the node feature matrix at time step t. After inserting graph convolution, the last equation becomes:

$$H^{(t)} = \sigma(Gconv(X^{(t)}, A; W) + Gconv(H^{(t-1)}, A; U) + b)$$

where $Gconv(\cdot)$ is a graph convolutional layer. Nevertheless, RNN-based methods suffer from time-consuming iterative propagation and gradient explosion/vanishing issues (Z. Wu et al., 2019).

As an alternative, CNN-based methods deal with spatial-temporal graphs in a non-recursive way. The advantages are that they can compute in parallel, gradients are stable, and memory required is lower than in RNN-based methods. Let's say that the inputs to a spatial-temporal graph neural network is a tensor $\chi \in R^{Tx\,n\,x\,d}$, the 1D-CNN layer slides over $\chi_{[:,i,;]}$ along the time axis to add temporal information for each node while the graph convolutional layer operates on $\chi_{[i,:,;]}$ to add spatial information at each time step.

4 Variational Autoencoders (VAEs)

In the recent years, Variational Autoencoders (VAEs) have become one of the most popular ways to unsupervised learning of complicated distributions. Unsupervised learning is a kind of deep learning that seeks for undetected patterns in a dataset with no pre-existing labels and with the minimal human supervision (Hinton & Sejnowski, 1999). When we talk about labels, we refer normally to the output the neural network is supposed to deliver. For example, in image recognition, a set of pixels that represents an image of a dog will be labelled as "dog". The neural network will output what it thinks the set of pixels is, and it will compare it to the ground truth label, in order to calculate the error and modify the parameters later.

As it has been explained in the last chapter, most of the machine learning models are "generative models". As a general rule, these models deal with distributions $P(X)$, defined over datapoints X in some potentially high-dimensional space χ. For instance, images are one popular type of data used in generative modelling. Each "datapoint" (image) has thousands or millions of dimensions (pixels), and the generative model's job is to capture the dependencies between those pixels: realistic shapes, similar colors between neighbor pixels, pixels organized into objects, etc. (Doersch, 2016). Usually, the objective of generative models is to produce more examples that are like those already in a database, but not exactly the same. For example, for a database of images we could recreate new, unseen images. One way to formalize this setup is by saying that we get objects X distributed according to some unknown distribution $P_{gt}(X)$, and the goal is to learn a model P which we can sample from, such that P is as similar as possible to P_{gt} (Doersch, 2016).

Figure 11: Generated image by a neural network (Source: (Joglekar, 2017))

However, training these generative models has suffered many problems during the years. There have been three serious drawbacks. First of all, they might require strong assumptions about the structure in the data. Second, they might make extreme approximations, bringing it to suboptimal models. Or third, they can rely on computationally expensive inference procedures like Markov Chain Monte Carlo. These drawbacks have been reduced in the years that have passed thanks to the use of backpropagation-based function approximators, since they helped a lot in training neural networks.

In the VAE, backpropagation is used to train the model in a fast way. Also, VAEs deal with the first drawback pretty well since the assumptions made on the models are weak. They also do some approximations, but the error introduced by them is relatively small (Doersch, 2016).

4.1 Theoretical description

A mathematical description of VAEs and the way they operate will be done in this chapter, according to the paper written by Carl Doersch, from the UC Berkeley.

4.1.1 Latent variable models

When a generative model is being trained, the more complicated the relations between the dimensions, the more difficult is to train. For example, when trying to generate images of handwritten digits 0-9, the model needs to decide some things. Let's say that if it wants to generate a 7, the left side cannot contain the right side of an 8, otherwise it will not look like any real digit. So, in an intuitive way, the model first decides which character is going to generate before start drawing it by assigning values to the pixels. This decision is formally called a *latent variable* (Doersch, 2016). Latent variables, in statistics, are variables that are not directly observed, they are inferred via a mathematical model from other variables that are measured and observed. They are used to reduce the dimensionality of data, trying to summarize an underlying concept.

There are many different techniques to infer latent variables: Hidden Markov Models, Factors analysis, Principal component analysis, Partial least squares regression, Latent semantic analysis, EM algorithms, Metropolis-Hastings algorithm, etc. Some of them have been used in the context of transport models, as mentioned in section 2.2.2.

In order for us to be allowed to say that our model is representative of the dataset, we need to make sura that for every datapoint X in the dataset, it exists one (or many) dispositions of the latent variables which let the model generate something very similar to X. Formalizing this

affirmation, say we have a vector of latent variables z in a high-dimensional space $\mathcal{Z}$ which we can easily sample according to some probability density function $P(z)$ defined over $\mathcal{Z}$. Then, say we have a set of deterministic functions $f(z;\ \theta)$, parametrized by a vector θ in some space Θ, where $f: \mathcal{Z}\ x\ \Theta\ \rightarrow\ \mathcal{X}$. f is deterministic, but if z is random and θ is fixed, then $f(z;\ \theta)$ is a random variable in the space $\mathcal{X}$. We want to optimize θ such that we can sample z from $P(z)$ and, with high probability, $f(z;\ \theta)$ will be like the X's in the studied dataset (Doersch, 2016).

To do so, we search to maximize the probability of each X in the training dataset under the entire generative process, according to the equation:

$$P(X) = \int P(X|z{:}\,\theta)P(z)\ dz$$

where $f(z;\ \theta)$ has been replaced by a distribution $P(X|z;\theta)$. The main idea behind this is the "maximum likelihood" criterion, which says that if the model is likely to produce training set samples, then it is also likely to produce similar samples, and unlikely to produce dissimilar ones. In VAEs, one usual way to choose this output distribution is a Gaussian one, i.e., $P(X|z;\theta) = \mathcal{N}(X|f(z;\ \theta), \sigma^2 * I)$. That is, it has mean $f(z;\ \theta)$ and covariance equal to the identity matrix I times some scalar σ. This is done like that in order to have some z which needs to result in samples that are just *like* X. Since we have a Gaussian distribution, gradient descent can be used to increase $P(X)$ by making $f(z;\ \theta)$approach X for some z. Note that the output distribution does not need to be Gaussian. The most important feature is simply that $P(X|z)$ can be computed, and is continuous in θ. From now on, we will omit θ from $f(z;\ \theta)$ to avoid confusion (Doersch, 2016).

4.1.2 The core of VAEs

The mathematical foundations of VAEs have relatively little to do with classical autoencoders. VAEs approximately maximize the already mentioned equation (equation 1 from now on):

$$P(X) = \int P(X|z{:}\,\theta)P(z)\ dz$$

The name "autoencoders" is taken because the final training objective derived from this setup contains an encoder and a decoder, and looks like a traditional autoencoder. There are two main issues that VAEs must deal with in order to solve equation 1: how to define the latent variables (which information they represent), and how to deal with the integral over z. VAEs provide a definite answer to both questions.

First of all, VAEs assume that it doesn't exist a simple interpretation of the dimensions of z, and instead affirm that samples of z can be drawn from a sample distribution, i.e., $\mathcal{N}(0, I)$, where I is the identity matrix. To achieve this, the point is to notice that any distribution in d dimensions can be generated by taking a set of d variables that are normally distributed and mapping them through a complicated enough function. Taking into account that:

$$P(X|z;\theta) = \mathcal{N}(X|f(z;\theta), \sigma^2 * I)$$

If $f(z;\theta)$ is a multi-layer neural network, then we can think of the network as using its first layers to map z's, which are normally distributed, to the latent values with exactly the appropriate statistics. Then it can use later layers to map those latent values to a complete output with the same shape like the input data.

Now we need to maximize equation 1, where $P(z) = \mathcal{N}(z|0, I)$. As it is usually done in machine learning, if it is possible to find a mathematical formula for $P(X)$, and we can have the gradient of that formula, then it is possible to optimize the model using stochastic gradient descent.

Remembering that the objective of VAEs is to produce data similar to the introduced one, there is a problem because without labels that indicate which datapoints are similar to each other, they are difficult to train. Instead, VAEs alter the sampling procedure to make it faster, without changing the similarity metric (Doersch, 2016).

In order to achieve the objective, it would be nice to have a shortcut when sampling to compute equation 1. In practice, for the majority of z, $P(X|z)$ will be so close to zero, and hence contribute almost nothing to the estimate of $P(X)$. The key is to try to sample values of z that which have a high probability of produce X, and compute $P(X)$ only from those. For this, we need another function $Q(z|X)$ which can take a value of X and supply a distribution over z values that are likely to produce X. Hopefully, the space of z values that are likely under Q will be much smaller than the space of all z's that are likely under the first $P(z)$. Now, we can compute $E_{Z\sim Q}P(X|z)$ quite easily. However, z is sampled from an arbitrary distribution with probability density function $Q(z)$, which is not $\mathcal{N}(0, I)$, so in order to optimize $P(X)$ we need to relate it to $E_{Z\sim Q}P(X|z)$.

To do so, we will define what is called the Kullback-Leibler divergence (KL divergence or $\mathcal{D}$) between $P(z|X)$ and $Q(z)$, for an arbitrary Q (which could or could not depend on X):

$$\mathcal{D}[Q(z)||P(z|X) = E_{Z\sim Q}[logQ(z) - logP(z|X)]$$

The KL divergence measures the difference from one probability distribution from a second one, which acts as the reference probability distribution. In the simplest case, if the Kullback-Leibler divergence is 0, it means that both probability distributions are identical.

We can introduce both $P(X)$ and $P(X|z)$ into this equation by applying Bayes rule to $P(z|X)$:

$$\mathcal{D}[Q(z)\|P(z|X) = E_{Z\sim Q}[logQ(z) - logP(X|z) - logP(z)] + logP(X)$$

Here, $logP(X)$ comes out of the expectation because it doesn't depend on z. After negating both sides, rearranging and contracting part of $E_{Z\sim Q}$ into a KL-divergence terms yields:

$$\log P(X) - \mathcal{D}[Q(z)\|P(z|X) = E_{Z\sim Q}[logP(X|z)] - \mathcal{D}[Q(z)\|P(z)]$$

Note that X is fixed, and Q can be any distribution. Since we're interested in inferring $P(X)$, it makes sense to construct a Q which does depend on X, and in particular, one which makes $\mathcal{D}[Q(z)\|P(z|X)$ small:

$$\log P(X) - \mathcal{D}[Q(z|X)\|P(z|X) = E_{Z\sim Q}[logP(X|z)] - \mathcal{D}[Q(z|X)\|P(z)]$$

This last equation is the core of the variational autoencoder. The left side has the quantity we want to maximize: $logP(X)$ (plus an error term, that makes Q produce z's that can reproduce a given X; this term will become small if Q is high capacity). The right part is something we can optimize via stochastic gradient descent given the appropriate choice of Q (Doersch, 2016).

Going deeply into the left side of the equation, we want to maximize $logP(X)$ while simultaneously minimize $\mathcal{D}[Q(z|X)\|P(z|X)]$. $P(z|X)$ is not something we can compute analytically, since it describes the values of z that are likely to produce a sample like X. However, the second term of the left side is pulling $Q(z|x)$ to match $P(z|X)$. Assuming we use an arbitrary high-capacity model for $Q(z|x)$, then $Q(z|x)$ will hopefully match $P(z|X)$, in which case this KL-divergence term will be zero, and we will be optimizing directly $logP(X)$.

How is it possible to carry out stochastic gradient descent on the right part of the aforementioned core equation? First, it is needed to know more specifically about the configuration that $Q(z|X)$ will take. Normally, $Q(z|X) = \mathcal{N}(z|\mu(X;\vartheta), \Sigma(X;\vartheta))$, where μ and Σ are again implemented via neural networks, and Σ is forced to be a diagonal matrix. The reason for this choice is merely computational, to make the calculations clearer. The final term $\mathcal{D}[Q(z|X)\|P(z)]$ results now in a KL-divergence between two multivariate Gaussian distributions (Doersch, 2016).

The first term on the right part of the core equation is a bit more complicated. We could use sampling methods to estimate $E_{Z\sim Q}[logP(X|z)]$, but to have a good estimation a considerable high number of samples would be needed of z through f, which would be an expensive process. As a result, as is standard in stochastic gradient descent, we take one sample of z and treat $P(X|z)$ for that z as an approximation of $E_{Z\sim Q}[logP(X|z)]$. The full equation to optimize now is:

$$E_{X\sim D}[logP(X) - \mathcal{D}[Q(z|X)\|P(z|X)]] = E_{X\sim D}[E_{Z\sim Q}[logP(X|z)] - \mathcal{D}[Q(z|X)\|P(z)]]$$

So, we can sample a single value of X and a single value of z from the distribution $Q(z|X)$, and calculate the gradient of:

$$logP(X|z) - \mathcal{D}[Q(z|X)\|P(z)]$$

We can then take an average of the gradient of this function over many samples of X and z taken arbitrarily.

There is a significant problem with the last equation though. $E_{Z\sim Q}[logP(X|z)]$ not only depends on the parameters of P, but also on the parameters of Q. But in the last equation this dependency has disappeared! In order to make VAEs work, it's very important to push Q to produce codes for X that P can successfully decode. The forward pass of the neural network works well and, if the output is averaged over many samples of X and z, produces the correct expected value. Nevertheless, the error needs to be back-propagated through a layer that samples z from $Q(z|X)$, which is a non-continuous operation and has no gradient.

To solve this, the so-called "reparameterization trick" is introduced, and it consists on moving the sampling layer to an input later. Given $\mu(X)$ and $\Sigma(X)$ – the mean and covariance of $Q(z|X)$ – we can sample from $\mathcal{N}(\mu(X), \Sigma(X))$ by first sampling $\epsilon \sim \mathcal{N}(0, I)$, then computing $z = \mu(X) + \Sigma^{1/2}(X) * \epsilon$. Thus, the equation we actually compute the gradient of is:

$$E_{X\sim D}[E_{\epsilon\sim\mathcal{N}(0,I)}\left[logP\left(X\middle|z = \mu(X) + \Sigma^{\frac{1}{2}}(X) * \epsilon\right)\right] - \mathcal{D}[Q(z|X)\|P(z)]]$$

This equation is shown schematically in Figure 12 (right).

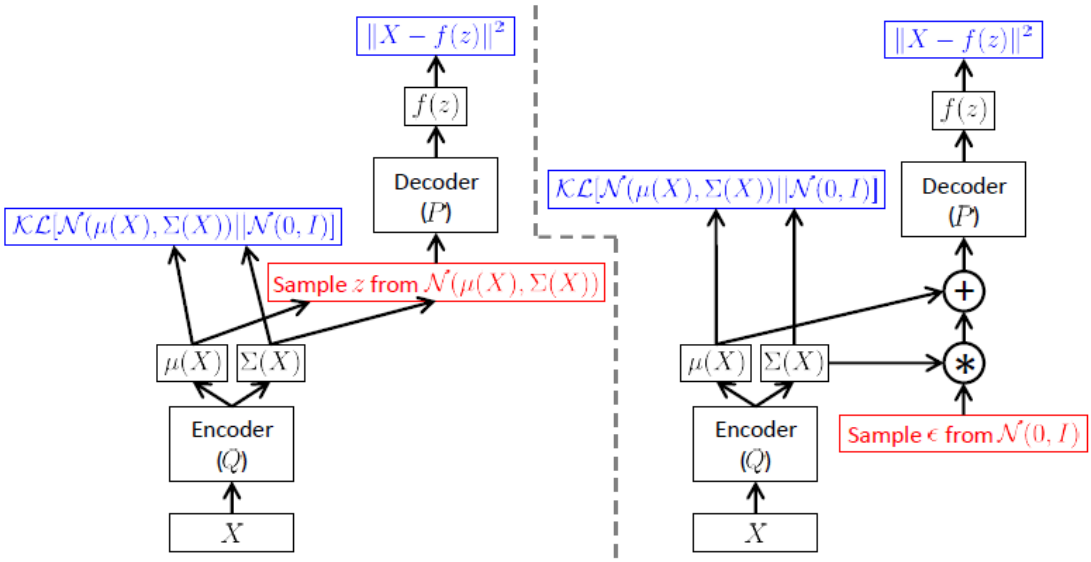

Figure 12: Schematic of a VAE. Left side is without the "reparameterization trick", and right with it (Source: (Doersch, 2016))

4.2 Example of application and inspiration

Now that it has been observed how a Variational Autoencoder works, let's see an example of what it can actually do with real world data. In machine learning, one of the common things is to work with a database of handwritten numerical digits, consisting of a series of images of 28x28 pixels with one color channel. This dataset is commonly named MNIST (Modified National Institute of Standards and Technology), and it contains 60.000 training images and 10.000 testing images.

The network will learn to reconstruct these digits and output them. First, an encoder will apply some convolutions to the input data, since it's made of images. Then, the same encoder will create two vectors containing data following a Gaussian distribution, which will be a vector of means, $\mu(X)$, and a vector of covariance, $\Sigma(X)$. These two vectors will be used to create the z-values from the latent space, which will be later fed to the decoder. The decoder does not care about whether the input values are sampled from a Gaussian distribution that has been created by us. It will simply try to reconstruct the input images, to this end, the network uses a series of convolutions again (Mohr, 2017). For computing the image reconstruction loss, the network uses squared difference. This loss is combined with the Kullback-Leibler divergence, which makes sure the latent values will be sampled from a normal distribution. After training the network, it is able to create new images of handwritten digits by simply sampling values from a

normal distribution and feeding them to the decoder. Surprisingly, most of the created digits look like if they were created by humans.

Figure 13: Examples of computer generated handwritten digits by a VAE (Source: (Mohr, 2017))

To this end, we have thought to apply this methodology to a transport dataset. As it will be explained in section 6.1 more in detail, the dataset used in this project consists on vectors of activity sequences of different individuals. If the VAE is able to learn a distribution and the main characteristics from a vector of pixels, then it should be able too to grasp the features of a vector of activity sequences. Certainly, the shapes and the information contained in the vectors are different, but after some adaptations are done, there should be no problem to encode the data to a latent space. And after encoding it, we should be able to create new sequences of activities just by sampling them from the latent space.

5 Long-Short Term Memory for sequence to sequence modelling

LSTM (Long Short-Term Memory) units are a kind of Recurrent neural networks (RNN), in fact, one of the most powerful and known subsets of them. RNNs are an artificial neural network which can be used to recognize data patterns in sequences, such as numerical time series data, stock markets prices, text, genomes, speeches, etc. What makes RNNs and LSTMs different from other neural networks is that they take into account time and sequence order, so they include a temporal dimension.

In the next sections, a theoretical description on RNNs will be done, based on a paper written by Sherstinsky (2018), which developed deeply the current literature by presenting the training formulas and justifying some concepts that had been presented before without complete justification. To avoid an excess of mathematical demonstrations, some intermediate parts of mathematical deductions will be omitted for the ease of the reader to follow the document.

5.1 Recurrent Neural Networks (RNNs)

5.1.1 The roots of RNN

In this section, Recurrent Neural Networks (RNNs) will be derived from differential equations. Let $\vec{s}(t)$ be the value of the d-dimensional state signal vector and consider the general nonlinear first-order non-homogeneous differential equation, which describes the evolution of the state signal as a function of time, t:

$$\frac{\mathrm{d}\vec{s}(t)}{dt} = \vec{f}(t) + \vec{\phi}$$

where $\vec{f}(t)$ is a d-dimensional vector-valued function of time, $t \in \mathbb{R}^+$, and $\vec{\phi}$ is a constant d-dimensional vector. One canonical form of $\vec{f}(t)$ is:

$$\vec{f}(t) = \vec{a}(t) + \vec{b}(t) + \vec{c}(t)$$

whose constituent terms, $\vec{a}(t)$, $\vec{b}(t)$, and $\vec{c}(t)$, are d-dimensional vector-valued functions of time, t. They are defined as follows:

$$\vec{a}(t) = \sum_{k=0}^{K_s-1} \vec{a}_k(\vec{s}(t-\tau_s(k)))$$

$$\vec{b}(t) = \sum_{k=0}^{K_r-1} \vec{b}_k(\vec{r}(t-\tau_r(k)))$$

$$\vec{r}(t-\tau_r(k)) = G(\vec{s}(t-\tau_r(k)))$$

$$\vec{c}(t) = \sum_{k=0}^{K_x-1} \vec{c}_k(\vec{x}(t-\tau_x(k)))$$

where $\vec{r}(t)$, the readout signal vector, is a warped version of the state signal vector, $\vec{s}(t)$. A common choice for the element-wise, nonlinear, saturating, and invertible "warping" (or "activation") function, $G(z)$, is an optionally scaled and/or shifter form of the hyperbolic tangent (Sherstinky, 2018).

Hence, in the resulting system,

$$\frac{\mathrm{d}\vec{s}(t)}{dt} = \sum_{k=0}^{K_s-1} \vec{a}_k(\vec{s}(t-\tau_s(k))) + \sum_{k=0}^{K_r-1} \vec{b}_k(\vec{r}(t-\tau_r(k))) + \sum_{k=0}^{K_x-1} \vec{c}_k\left(\vec{x}(t-\tau_x(k))\right) + \vec{\phi}$$

the time rate of change of the state signal depends on three main parts plus the bias term $\vec{\phi}$. The first "analog" component $\sum_{k=0}^{K_s-1} \vec{a}_k(\vec{s}(t-\tau_s(k)))$, is the combination of up to K_s time-shifted (by the delay time constants, $\tau_s(k)$) functions, $\vec{a}_k(\vec{s}(t))$, where the term "analog" underscores the fact that each $\vec{a}_k(\vec{s}(t))$ is a function of the (maybe shifted) state signal *per se*.

The second component $\sum_{k=0}^{K_r-1} \vec{b}_k(\vec{r}(t-\tau_r(k)))$, is the combination of up to K_r time-shifted (by the delay time constants, $\tau_r(k)$) functions, $\vec{b}_k(\vec{r}(t))$, of the readout signal, the warped (binary-valued in the extreme) version of the state signal.

And the third component, $\sum_{k=0}^{K_x-1} \vec{c}_k\left(\vec{x}(t-\tau_x(k))\right)$, represents the external input, composed of the combination of up to K_x time-shifted (by the delay time constants, $\tau_x(k)$) functions, $\vec{c}_k(\vec{x}(t))$, of the input signal.

The reason of selecting a form of the hyperbolic tangent as the activation function relies in the possession of some useful properties. First, it is monotonic and negative-symmetric with a quasi-linear region, whose slope can be regulated (Metropolis et al., 1953). On the other hand, it is bipolarly-saturating (i.e. bonded at both the negative and the positive limits of its domain).

The quasi-linear mode aides in the design of the system's parameters and in interpreting its behavior in the "small signal" regime (i.e. when $\|\vec{s}(t)\| \ll 1$). The bipolarly-saturating ("squashing") aspect, together with the proper design of the internal parameters of the functions $\vec{a}_k(\vec{s}(t))$ and $\vec{b}_k(\vec{r}(t))$, helps to keep the state of the system (and thus, the output) bounded.

Separately, the time delay terms on the right-hand side of the equation contain the "memory" aspects of the system. They enable the quantity holding the instantaneous time rate of change of the state signal $\frac{\mathrm{d}\vec{s}(t)}{dt}$, to incorporate contributions from the state, the readout, and the input signal values, measured at different points in time, relative to the current time, t (Sherstinky, 2018).

Since time is an important factor in this networks, to accommodate "Back Propagation of Error" to RNNs, both continuous-time and discrete-time versions of "Back Propagation Through Time" (BPTT) have been developed and used to train the weights and time delays of these networks (Elman, 1990; Jordan, 1986; Pearlmutter, 1989; Pineda, 1987). We will rely on BPTT for training the systems analyzed in this section.

Let's now suppose that $\vec{a}_k(\vec{s}(t-\tau_s(k)))$, $\vec{b}_k(\vec{r}(t-\tau_r(k)))$, and $\vec{c}_k\left(\vec{x}(t-\tau_x(k))\right)$ are linear functions of $\vec{s}$, $\vec{r}$, and $\vec{x}$, respectively. Then, the last equation become a nonlinear delay differential equation (DDE) with linear coefficients.

$$\frac{\mathrm{d}\vec{s}(t)}{dt} = \sum_{k=0}^{K_s-1} A_k(\vec{s}(t-\tau_s(k))) + \sum_{k=0}^{K_r-1} B_k(\vec{r}(t-\tau_r(k))) + \sum_{k=0}^{K_x-1} C_k\left(\vec{x}(t-\tau_x(k))\right) + \vec{\phi}$$

After making some calculations, rearranging and making some simplifications, the canonical Recurrent Neural Network (RNN) form results in (Sherstinky, 2018):

$$\vec{s}[n] = W_s\vec{s}[n-1] + W_r\vec{r}[n-1] + W_x\vec{x}[n] + \vec{\theta}_s$$

$$\vec{r}[n] = G(\vec{s}[n])$$

where $W_s = (I - (\Delta T)A)^{-1}$, with ΔT being the sampling time step and $A = A_0$, $W_r = (\Delta T)W_sB$, with $B = B_0$, $W_x = (\Delta T)W_sC$, with $C = C_0$, and $\vec{\theta}_s = (\Delta T)W_s\vec{\phi}$.

The diagram of this equation can be seen in Figure 14:

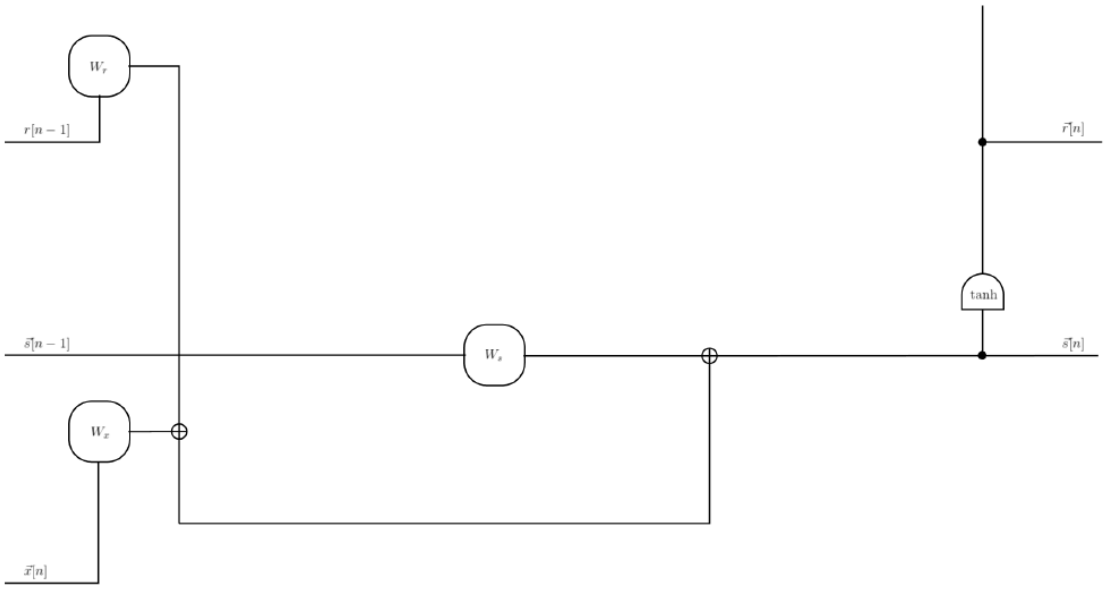

Figure 14: Canonical RNN cell. The bias parameters $\vec{\theta}_s$, have been omitted for brevity. It can be assumed to be included without the loss of generality by appending an additional element, always set to 1, to the input signal vector, $\vec{x}[n]$, and increasing the row dimensions of W_x by 1. (Source: (Sherstinky, 2018))

For the system in the canonical equation to be stable, every eigenvalue of $\widehat{W} = W_s + W_r$ must be within the complex-valued unit circle. Since there is considerable flexibility in the choice of the elements of A and B to satisfy this requirement, setting the sampling time step $\Delta T = 1$ for simplicity is acceptable. As another simplification, let A be a diagonal matrix with large negative entries on its main diagonal. Then, $W_s \approx -A^{-1}$ will be a diagonal matrix with small positive entries, $\frac{1}{|a_{ii}|}$, on its main diagonal, which means that the explicit effect of the state signal's value from memory, $\vec{s}[n-1]$, on the system's trajectory will be negligible. Thus, ignoring the first term of the equation, reduces it to the standard RNN definition:

$$\vec{s}[n] = W_r\vec{r}[n-1] + W_x\vec{x}[n] + \vec{\theta}_s$$

$$\vec{r}[n] = G(\vec{s}[n])$$

Now, only the matrix $\widehat{W} \approx W_r \approx -A^{-1}B$ is responsible for the stability of the RNN. However, stability considerations will be later revisited in order to justify the need to evolve the RNN to a more complex system, the LSTM.

5.1.2 RNN Unfolding/Unrolling

Is it convenient to use the term "cell" when referring to the last equation in the uninitialized state. In other words, the sequence has been defined by these equations, but its terms not yet computed. Then the cell can be said to be "unfolded" or "unrolled" by specifying the initial

conditions on the state signal, $\vec{s}[n]$, and numerically evaluating the equation for a range of discrete steps, indexed by n. This process is illustrated in Figure 15 below.

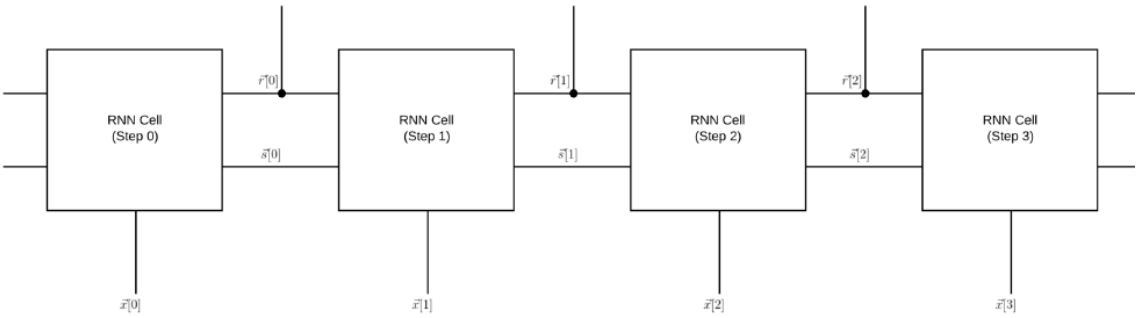

Figure 15: Sequence of steps generated by unrolling an RNN cell (Source: (Sherstinky, 2018)).

The RNN equation is recursive in the state signal, $\vec{s}[n]$. Hence, due to the repeated application of the recurrence relation as part of the unrolling, the state signal, $\vec{s}[n]$, at some value of the index, n, no matter how large, encompasses the contributions of the state signal, $\vec{s}[k]$, and the input signal, $\vec{x}[k]$, for all indices, $k < n$, ending at $k = 0$, the start of sequence (Elman, 1990; Jordan, 1986).

5.1.3 RNN training difficulties

Once the infinite RNN sequence is truncated (or unrolled to a finite length), the resulting system becomes inherently stable. However, RNN systems are problematic in practice, despite their stability. During training, they struggle with the well-documented problems of "vanishing gradients" and "exploding gradients" (Hochreiter & Schmidhuber, 1997; Pascanu et al., 2013). Truncated unrolled RNN systems, are commonly trained using BPTT, which is the "Back Propagation" technique adapted for sequences. Originally, Back Propagation was restricted to feedforward networks only. Subsequently, it has been successfully applied to recurrent networks by taking advantage of the fact that for every recurrent network there exists an equivalent feedforward network with identical behavior for a finite number of steps (Sherstinky, 2018). As a supervised training algorithm, BPTT uses the available $\vec{x}_m[n]$ and $\vec{r}[n]$ data pairs in the training set to compute the parameters of the system, $\Theta \equiv \{W_r, W_x, \vec{\theta}_s\}$, so as to optimize an objective function (i.e. the error function), E, which depends on the readout signal $\vec{r}[n]$, at one or more values of the index, n. If Gradient Descent is used to optimize E, then BPTT provides a consistent procedure for deriving the elements of $\frac{\partial E}{\partial \Theta}$ through a repeated application of the chain rule.

The problem comes when using the chain rule itself. In a network of n hidden layers, n derivatives will be multiplied together. If these derivatives are small, the gradient will start decreasing exponentially as the error propagates through the model, until it eventually vanishes, hence the "vanishing gradient" problem. On the other hand, if the derivatives are large, then the gradient increases exponentially while propagating back through the model until it eventually explodes, in terms of computational capacity.

The consequences of a vanishing gradient, is that the model is incapable of learning meaningful insights, since the weights and biases of the initial layers, which tend to learn the core features from the input data, will not be updated effectively. In the worst case, if the gradient is 0, the network will stop training since there is no direction to update.

Alternatively, the problems of exploding gradients is that the model is very unstable and incapable of effective learning. Weights and biases are changed drastically in every step, which at some point become so large that cause an overflow causing NaN values that the computer can no longer update (Pykes, 2020). The most effective solution so far is the Long Short-Term Memory (LSTM) cell architecture (Graves, 2008; Hochreiter & Schmidhuber, 1997; Pascanu et al., 2013).

5.2 The Long Short-Term Memory network

Long Short-Term Memory networks were introduced by (Hochreiter & Schmidhuber, 1997) and have been later deeply developed. They are specially designed to overcome the problem of the long-term dependency, since they are very capable of remembering information for long periods of time.

For example, consider the case of an activity-based model, where we have a sequence of all the activities a person carries out during a day. If we are trying to predict the activity that the person will be doing at the end of the day, it will probably be being at home, no matter how confusing or large the information on previous activities is. In this case, the gap between the previous relevant information and the place that it's needed is small, so RNNs can do this work. But for example, consider trying to predict the activity someone will be doing at 19:00 h. If the person has gone from home to work, traveling in between, then he/she has gone eating outside, etc. and other sets of activities, but for example, shopping is not in between this set of previous activities, it might be highly probable that this person might stop somewhere to shop. In this case, information about previous activities is important, and the furthest this information goes,

the better. If we already know that the person has gone shopping, it is more unlikely he/she will go shopping again that day, and so on. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large. RNNs are not capable to solve this, but fortunately, LSTMs can.

All RNNs have the form of a chain repeating cells as it can be seen in Figure 15. In standard RNNs, these cells have a very simple structure only with a single hyperbolic tangent layer (Figure 14).

LSTMs are organized also in a chain-like structure, but the repeating cell is different on the inside. Instead of having a single neural network layer, there are four, interacting in a quite different and interesting way.

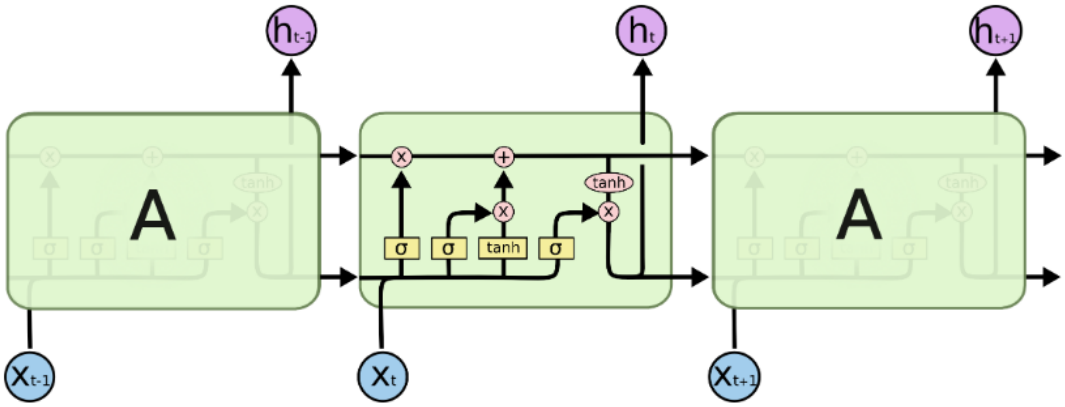

Figure 16: Three unrolled LSTM cells with the internal structure of one cell (Source: (Olah, 2015))

The notation used in the last diagram is the following:

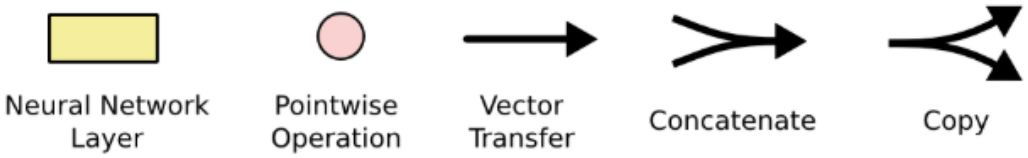

Figure 17: Notation for Figure 16 (Source: (Olah, 2015))

In the diagram, each line is an entire vector, from the output of one cell to the input of the following one. The pink circles are pointwise operations, for example vector addition, while the yellow boxes are learned neural network layers (Olah, 2015).

The most important idea in LSTMs is the cell state $\vec{C}_t$ (state signal, $\vec{s}[n]$, in the RNN section), the horizontal line running through the top of the entire diagram. This cell state operates as a

conveyor belt, running down the whole chain with small linear interactions. It's very easy for information flows to just go along being unchanged.

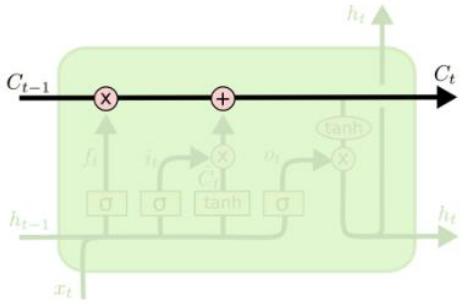

Figure 18: The cell state vector line (Source: (Olah, 2015))

The LSTM has the ability to add or remove information to the cell state, a process that is regulated by structures that operate as gates. These gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

The sigmoidal (also known as "logistic") nonlinearity is a good choice, because it is bipolarly-saturating between the values 0 and 1 and is monotonic, continuous, and differentiable. If the value of the sigmoid layer is zero, the gate won't let nothing through. Otherwise, if it's one it will let all the information through.

The LSTM has three gates, to protect and control the information contained in the cell state (Olah, 2015).

To begin with, the LSTM has to decide which information is it going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate", since it removes information from the memory. This gate looks at the previous hidden state, h_{t-1}, and at the input signal, x_t, at time t, and delivers an output between 0 and 1 for each number in the cell state C_{t-1}. As mentioned before, a 1 means that the cell has to completely keep this information, and a 0 means that has to completely get rid of it.

Let's go back to the example of trying to predict the next activities based on the previous ones. In such a case, the cell state might include the activity of being at home, when it sees a new activity, such as traveling, we want to forget that the person is at home for the new activity, which is traveling, since traveling will certainly develop in doing a different activity than the prior one.

The equation of the "forget gate" is formalized as:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

where σ represents the sigmoid function, W_f the matrix of weights of the neural network, and b_f the biases matrix of the neural network.

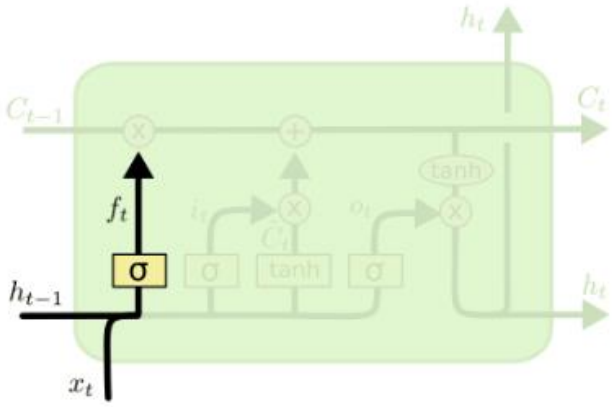

Figure 19: Structure of the "forget gate" in the LSTM cell (Source: (Olah, 2015))

Once it has been decided what will be forgotten and what will be kept, the next step is to decide what new information is it going to be added in the cell state. This is done in a two-step process. First, a sigmoid layer called the "input gate layer" takes the decision of which values will be updated. To continue, a hyperbolic tangent layer creates a vector of new candidate values, $\overrightarrow{\tilde{C}_t}$, that could be added to the cell state. Then, they are combined to create an update to the state (Olah, 2015).

In the example of the activities, we'd like to add the new activity, traveling, to the cell state, to replace the old one, being at home, we're forgetting.

The mathematical expression for the "input gate" is (vector arrows will be omitted for simplicity):

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

where σ represents the sigmoid function, W_i and W_C the matrix of weights of the sigmoidal neural network and the hyperbolic tangent neural network, respectively, and b_i and b_C the biases matrix of the sigmoidal and the hyperbolic tangent neural networks, respectively.

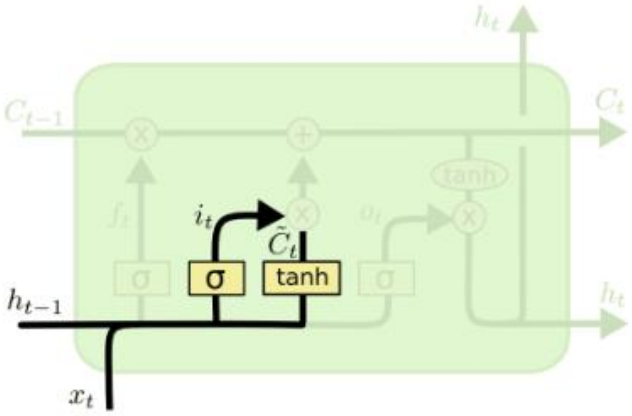

Figure 20: Structure of the "input gate" in the LSTM cell (Source: (Olah, 2015))

Now we need to update the old cell state, C_{t-1}, into the new one, C_t. The previous steps have already decided what to do, now we just need to effectively do it.

First of all, the old state is multiplied by the output of the "forget gate", f_t, forgetting the information we decided to forget earlier. Then $i_t \cdot \tilde{C}_t$ it's added. This is the new candidate values, scaled by how much we want to update each state value (Olah, 2015).

In the transport/activity case, we would drop the information that the person is at home and add the new activity, as we decided in the previous steps.

So, the new cell state results as:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

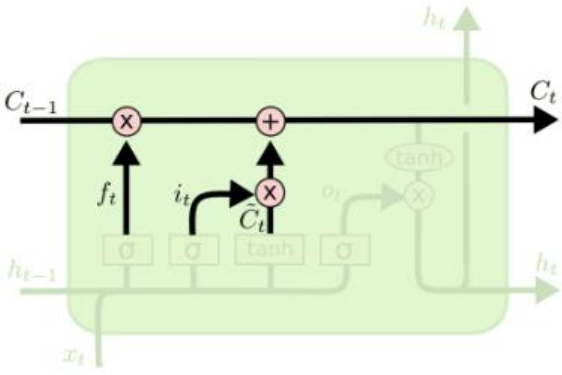

Figure 21: Updating the new cell state in the LSTM cell (Source: (Olah, 2015))

Finally, it needs to be decided what the output is going to be. This output will be based on the cell state, but with a filtered version. First of all, a sigmoid layer is used, which decides what parts of the cell state are going to be outputted. Them, the cell state, C_t, is run through a

hyperbolic tangent (to push the values to be between -1 and 1) and multiply the result by the output of the mentioned sigmoid gate, so that only the parts that we decided to output are actually taken (Olah, 2015).

The formalized equation results:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t \cdot \tanh(C_t)$$

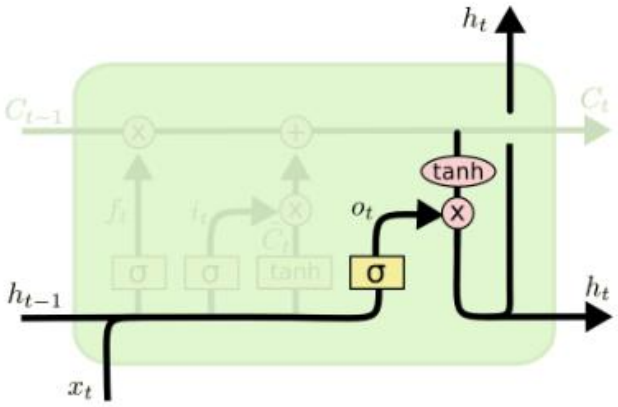

Figure 22: Structure of the "output gate" in the LSTM cell (Source: (Olah, 2015))

In the transport example, since it just saw a traveling activity, it might want to output information relevant to a being to work activity, in case that's the activity coming next.

5.3 Example of application and inspiration

One of the examples that served as an inspiration for this project is a model based on an encoder-decoder system that provides a pattern for using LSTMs to address a sequence-to-sequence prediction problem, such as machine translation (Brownlee, 2017b). Sequence-to-sequence learning (Seq2Seq) is about training neural networks in order to convert sentences from one domain (e.g. sentences in English) to sequences in another domain (e.g. the same sentences translated to French) (Chollet, 2017).

If the input sequence and the output one don't have the same length, the entire input sequence is needed in order to start predicting the target. For this reason, a RNN layer such as a LSTM (or several LSTMs unrolled) acts as an encoder, processing the input sequence and returning an internal state. This state serves as the "context" for the decoder that comes next, which is also a LSTM (or several unrolled) trained to predict the next characters of the target sequence, given previous characters of the same sentence. More specifically, it turns the target sequences into

the same ones but offset by one timestep in the future, a training process called “teacher forcing” (Chollet, 2017).

In inference mode, i.e. when we want to decode unknown input sequences, the process is slightly different:

1) Encode the input sentence into state vectors in the latent space.
2) Start with a target sequence of size one (just the start-of-sequence character, which can be a “_” character).
3) Feed the state vectors (h_t and c_t) and the one-character target sequence to the decoder again, in a recursive way, to get the predictions for the next character.
4) Sample the next character with the last predictions (using a “SoftMax” function).
5) Add the sampled character to the target sequence
6) Repeat from point 3 until the end-of-sequence character is generated or the character limit is reached.

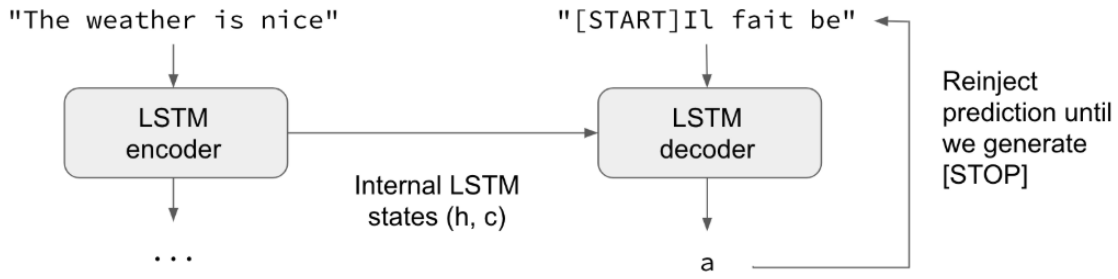

Figure 23: Inference mode for seq2seq prediction from English sentences to French sentences (Source: (Chollet, 2017))

Following this model, we have thought that since we have sequences of activities during a whole day, we could use the information on the first part of the sequence or X timesteps to predict the next Y timesteps or the whole last part of the sequence. For example, knowing what activities the person has done from 00:00 h to 12:00 h, try to predict what activities he/she will do during the rest of the day (i.e. from 12:00 h to 23:59 h) or for a specific time slot (e.g. from 12:00 h to 15:00 h). To do that, we would train the network using “teacher forcing” and then use the inference mode similarly to the machine translation case to generate the next sequence of activities.

6 Modelling an activity-based transport dataset

As mentioned in the beginning of this work, the aim of the study is to try to model transport phenomena from a population dataset. As it has been seen in section 2.2.2, this is a task that has been done more or less successfully in the past with diverse methods. However, not much approaches have addressed this issue with the perspective of machine learning with neural networks.

Since it is a quite new approach, there is not a lot of literature on how to better address the topic. Nevertheless, in this line, Karlaftis and Vlahogianni (2011) investigated the performance of neural networks versus the classical statistical methods used in transportation research. They found out that neural networks usually perform better when we try to model complex datasets, since they are more flexible and able to deal with nonlinearities and missing data. The most important machine learning methods include K-nearest neighbor (Cai et al., 2016), support vector regression (Asif et al., 2013), etc. Also, nonparametric approaches, such as Kalman filters (Chien et al., 2003), and matrix/tensor factorization methods (Tan et al., 2016) are also very used in problems of predicting traffic. Deep learning models, as a branch of machine learning models, have become popular and used in the traffic forecasting area (Cui et al., 2020). Most of the new models proposed for traffic forecasting (Ma et al., 2015; Yuankai Wu & Tan, 2016; H. Yu et al., 2017; Zhao et al., 2017) are based on RNNs and its improved version, the LSTMs. The mainly deal with sequence data by keeping a chain-like structure and internal memory with loops. To deal with the problem of missing data in LSTMs, Cui et al. (2020) proposed a model which can improve traffic prediction accuracy and robustness. They also investigated how to capture dependencies of traffic states series in a reverse chronological order, which was something that had been not explored before. Also, Wei et al. (2019) proposed a model which combined an Autoencoder with a LSTM network. The Autoencoder was used to capture the internal relationships of traffic flow, and with this acquired data and the historical one, the LSTM network predicted complex linear traffic flow data. All of these RNNs based models, generally outperformed the current traffic forecasting systems implemented in the cities or regions where the data was taken from. On a different approach, Ma et al. (2017) proposed a “convolutional neural network (CNN)-based method that learns traffic as images and predicts large-scale, network-wide traffic speed with high accuracy”. The spatiotemporal dynamics of traffic are

converted to images from where the CNN extracts traffic features and network-wide traffic speed prediction. In summary, since traffic flow and speed prediction is a complicated task due to the stochastic nature of the data, deep learning methods based on neural networks are being implemented more and more recently, due to their ability to mine big data and discover internal structures and potential features that the classical models are not able to discover. Nevertheless, most of the studies focus on traffic flow prediction or in traffic speed prediction, but few of them address the topic via the activity schedules derived from trip diaries.

As the object of this study, we will model a dataset provided by the promotors of this study, which contains information about activity schedules from the Belgian population. From the previously analyzed models of neural networks, the latter two different approaches will be used. The first one consists on a Variational Autoencoder (VAE). The second one consists in an encoder-decoder architecture based on Long-Short Term Memory (LSTM) cells.

After a brief description of the dataset, both models of neural networks will be explained and we will analyze the obtained results.

6.1 The Belgium Daily Mobility (BELDAM) dataset

The Belgian National Household Survey BELDAM (Belgium Daily Mobility) was carried out from December 2009 to December 2010 and portrayed the mobility behavior of Belgians. In total information of 8,532 household (i.e. 15,821 people aged 6 and over) were collected (Cornelis et al., 2010).

The dataset used in this thesis stems from the data obtained in this BELDAM project. The data was filtered and restructured to serve the purposes of the thesis. The dataset consists of activity-travel diary data of 11.302 individuals, primarily displaying the different activities they were doing at each time step. Data was structured in time steps are of 2 minutes, having a total of 720 time step values for each individual (720 time steps of 2 minutes are equivalent to a full day, 24h). Thus, for each individual we have a vector of 720 values, each value representing the activity he/she was doing at the considered time.

The activities are classified in 13 different types, and include the vast majority of things that a person can do during a day.

Activity ID	Activity definition
1	drop off / look for someone
2	being at home
3	being at work
4	work related displacement
5	follow a course (school, university, etc.)
6	have a meal outside
7	go to the supermarket/shopping
8	services (doctor, bank, etc.)
9	visiting family or friends
10	take a walk, do a tour
11	leisure, sports, culture
12	others
13	traveling

Table 1: Activity types considered in this project

6.2 Generating activity schedules

As it has been said in previous sections, transport demand modelling is a challenging task but necessary if we want to develop a good transport organization which solves the problems of the current situation and delivers new solutions for the future.

With the provided BELDAM dataset, which includes activity sequences from a set of individuals during a whole day, we will try to grasp the main features of this population in order to get metrics that can be useful for future transport modelling.

In this direction, we want to generate a new set of population based on the data obtained by the BELDAM study, and see how different or similar is this generated population from the original one. To do so, we will propose two different models. The first one will be a simplistic model based on a frequency analysis of the population, which is a fast and easy way to catch the main features of the set. The second one will be a model based on a Variational Autoencoder, which is more tedious and slower to calculate, but it should deliver better results.

Finally, both models will be compared according to some pre-established metrics, which are the following (all the metrics are referred per one day):

- Number of trips carried out by an individual

- Percentage of hours spent traveling by an individual
- Number of activities carried out outside from home by an individual
- Percentage of hours that a person spends outside from home
- Percentage of hours that a person spends at work
- Percentage of hours that a person spends at home
- Percentage of hours that a person spends at work trips
- Percentage of hours that a person spends following courses
- Percentage of hours that a person spends having a meal outside
- Percentage of hours that a person spends shopping
- Percentage of hours that a person spends taking services
- Percentage of hours that a person spends visiting family or friends
- Percentage of hours that a person spends walking or making a tour
- Percentage of hours that a person spends in leisure, culture or sports

Once both models have been analyzed, a comparison will be done to show which performs better when trying to generate a population as similar as possible to the original one.

6.2.1 Generation with a Frequency analysis of population

Frequency analysis consists in the study of the frequency of data in a group or set. It is very used, for example, in cryptanalysis, where they study the frequency of letters in a text. In human languages, some letters appear more than others, and this changes with the language. In English, for example, E, T, A, and O are the most common letters while Z, Q, X and J are rare. In a similar way happens in activity-based models. From our set of activities, we could say that some are more common, like being at home (2), going to work (3), or following courses (5), and some of them occur more sporadically, like doing work related trips (4) or having a meal outside from home (6).

The idea is to capture the frequencies of these activities during the day and try to generate a synthetic population that replicates the model.

6.2.1.1 Description of the model

The model is quite simple. The main idea is to analyze the frequency of each activity in each timestep of 2 minutes and from these probabilities, generate a new model by sampling data from this probability distribution. For example, if at 11:00 h, probabilities are of 60% that people are at work, 20% that they are at home, 10% that they are traveling and so on until 100%, we

would generate a random sample based on this vector of probabilities for the 11:00 h timestep. We have generated the same number of samples that we will generate with the VAE, in order to have comparable results.

6.2.1.2 Results

After having generated the population we will compute the means of each one of the aforementioned metrics. The analyzed sample consists of 1375 generated individuals. In summary, the mean for each metric of the Frequency Analysis (FA) can be found in Table 2. At the rightest column, we can find the computed values of the original population from the BELDAM dataset. We have to say that in this table, the values have been computed with a shifted version of the whole dataset, where we have eliminated sequences from weekends and from people older than 65 years old. The reason for this is because we thought that eliminating those variables, data will have less variability and results would be more accurate. However, later we discovered that this was not what was happening, but the contrary. Nonetheless, for comparative purposes like the table below, if we calculate the results with the same dataset, there is no problem.

Metric	Value (FA)	Value (BELDAM)
Number of trips carried out by an individual	35,67	3,249
Number of activities carried out outside from home by an individual	10,85	1,564
Percentage of hours spent traveling by an individual	5,477 %	5,641 %
Percentage of hours that a person spends outside from home	32,87 %	33,45 %
Percentage of hours that a person spends at work	13,93 %	14,08 %
Percentage of hours that a person spends at home	67,13 %	66,55 %
Percentage of hours that a person spends at work trips	0,9798 %	0,9722 %
Percentage of hours that a person spends following courses	5,053 %	6,217 %
Percentage of hours that a person spends having a meal outside	0,3918 %	0,298 %
Percentage of hours that a person spends shopping	1,574 %	1,228 %
Percentage of hours that a person spends taking services	0,3843 %	0,4754 %
Percentage of hours that a person spends visiting family or friends	1,764 %	1,557 %

Metric	Value (FA)	Value (BELDAM)
Percentage of hours that a person spends walking or making a tour	0,7716 %	0,3003 %
Percentage of hours that a person spends in leisure, culture or sports	1.143 %	1,283 %

Table 2: Value of the proposed metrics for the Frequency analysis model

As it can be seen in the table, the model is quite capable to reproduce with reliability the percentages of hours that individuals spend in each activity in an aggregate level. In some way, this is perfectly understandable, taking into account the way the sequences were generated. The generation method is based on the probabilities of doing each activity in each timestep, and for this reason, the outputted percentages are quite similar to the original ones, with small variations linked to the sampling randomness.

However, we can see that in the first two metrics, "number of daily trips carried out by an individual" and "number of daily activities carried out outside from home by an individual", the results obtained are very different from the original data. Why does this happen? To find the answer, we have to look at the data at a disaggregate level.

If we check out one sequence generated by this FA method, we will see that it lacks coherence. It lacks coherence both in the travel patterns and in the activity sequences.

In the travel patterns, we can see that there is not an alternation between a number different from 13 (any of the activities) and a 13 (traveling). In the original data, there is always a number 13 between two different activities, which indicates that the person made a trip to move from one activity and/or place to the following one. In the FA model, we can find some generated sequences where the person is at home from 17:08 h to 17:16 h. Then at 17:18 h he/she is suddenly following a course, at 17:20 h the individual is at work, at 17:22 h he/she is shopping, at 17:24 h he/she is following a course again, etc. Of course, this sequence is not humanly possible, since from one activity to the other one there's always has to be some traveling in between. In other words, sequences are not subjected to time and space constraints, since the person moves from one activity to the next one with no travel in between and in a considerably short amount of time.

The previous example also serves to see that the activities don't follow a coherent sequence. Nobody goes to follow a course for 2 minutes and then he/she is at work, and 2 minutes after

having entered to work he/she is suddenly shopping. Activities use to last for a longer period of time than a 2 minutes timestep. Also, there are some activities which normally precede others, for example going to work after being at home (always with a trip in between). These dependencies are not captured neither by the FA model.

In summary, the model is able to reproduce the percentage of activities that each person does every day, but it's very bad at the time of ordering those sequences and at the time of generating trips, since it is not able to caption time and space constraints and neither the usual relationships of precedence and posteriority between activities.

6.2.2 Generation with VAE

6.2.2.1 Description of the model

In this section, we will describe how the implementation of the Variational Autoencoder has been done in practice. The set-up has been done in a Python language environment. The reason for this choice is that there exists a high number of libraries and utilities in Python to develop machine learning models, and for now it is the most common language to program neural networks, and thus, there is a lot of information and references and trouble-shooting on the internet. Two of the most important frameworks that have been used to implement the model, and that are the most commonly used when it comes to machine learning, are "Tensorflow" and "Keras". "Tensorflow" was created by the Google Brain team, and it is an open source library for large-scale machine learning problems and numerical computation. On the other hand, "Keras" is a high-level neural networks library created by François Chollet that runs on the top of "Tensorflow".

To begin with, a sampling layer has been defined. As it has been said in section 4.1.2, this layer samples z from $z = \mu(X) + \Sigma^{\frac{1}{2}}(X) * \epsilon$ by using the "reparameterization trick".

```
class Sampling(layers.Layer):
  """Uses (z_mean, z_log_var) to sample z, the vector encoding a
sequence."""

  def call(self, inputs):
    z_mean, z_log_var = inputs
    batch = tf.shape(z_mean)[0]
    dim = tf.shape(z_mean)[1]
    epsilon = tf.keras.backend.random_normal(shape=(batch, dim))
    return z_mean + tf.exp(0.5 * z_log_var) * epsilon
```

To continue, the encoder is defined. The first part of the encoder is an input layer, which allows the network to create a total number of nodes equal to the dimension of the input data. In our

case, since we have sequences of 720 timesteps, the first dimension of the input layer is that number. The second dimension is 14, which is the dimension of the data (13 activity types) plus one.

Following this layer, two 1-dimensional convolutional layers are introduced. Convolution between two functions (f and g) produces a third function $(f \cdot g)$ that expresses how the shape of one is modified by the other. An example of 1-dimensional convolution can be seen in Figure 24. These convolutional layers are activated by a rectified linear unit (ReLU) function. We can also see two additional parameters called padding and stride. In a convolution, we can observe that the size of the output data is smaller than the input data. To keep the dimension the same, we use padding, which consists on adding zeros to the input matrix symmetrically. If the padding parameter is set to "same" like in our model, it will add the padding required to the input data to generate an output that has the same shape than the input (Brownlee, 2019b). Stride refers to the number of steps that we are moving at each convolution, and is set to one, which is the default number, without making jumps, just going through all the data.

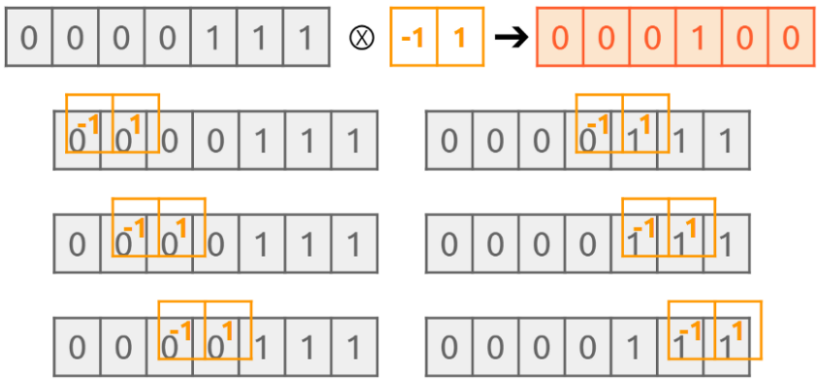

Figure 24: Convolution between a 1-diemensional vector of shape (1,7) and a 1-dimensional vector of shape (1,2) (Source: (Jeong, 2019))

Then, one "Flatten" layer is added, which converts a matrix of nodes into a single vector, so from a Conv1D layer of 720x32 dimension, we will go to a layer of dimension 23.040. Flattening is used to convert the data into a 1-dimensional array for inputting it to the next layer. We flatten the output of the convolutional layers to create a single long feature vector (Jeong, 2019).

Connected to this layer, it comes a fully-connected layer called "Dense" in the "Keras" framework. When we refer to fully-connected layers, we talk about all nodes from the previous layer being connected with an edge to all the nodes of the current layer. A combination of flattening and fully-connected layers emerging from a matrix of data can be seen in Figure 25.

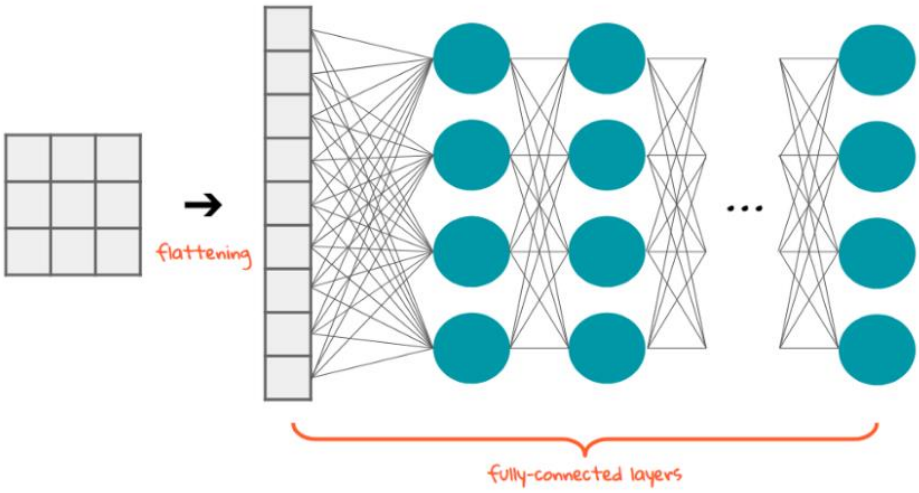

Figure 25: A (3x3 → 9x1) "Flatten" layer connected to several "Dense" layers of 4 nodes (Source: (Jeong, 2019))

The last two "Dense" layers are assigned to the mean and the covariance of $Q(z|X)$, $\mu(X)$ and $\Sigma(X)$, respectively, which will be used in the "Sampling" layer that comes next. The complete code for the encoder is as follows:

```
encoder_inputs = keras.Input(shape=(720, 14))
x = layers.Conv1D(64, 3, activation='relu', strides=1,
padding='same')(encoder_inputs)
x = layers.Conv1D(32, 3, activation='relu', strides=1,
padding='same')(x)
x = layers.Flatten()(x)
x = layers.Dense(32, activation='relu')(x)
z_mean = layers.Dense(latent_dim, name='z_mean')(x)
z_log_var = layers.Dense(latent_dim, name='z_log_var')(x)
z = Sampling()([z_mean, z_log_var])
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var, z],
name='encoder')
encoder.summary()
```

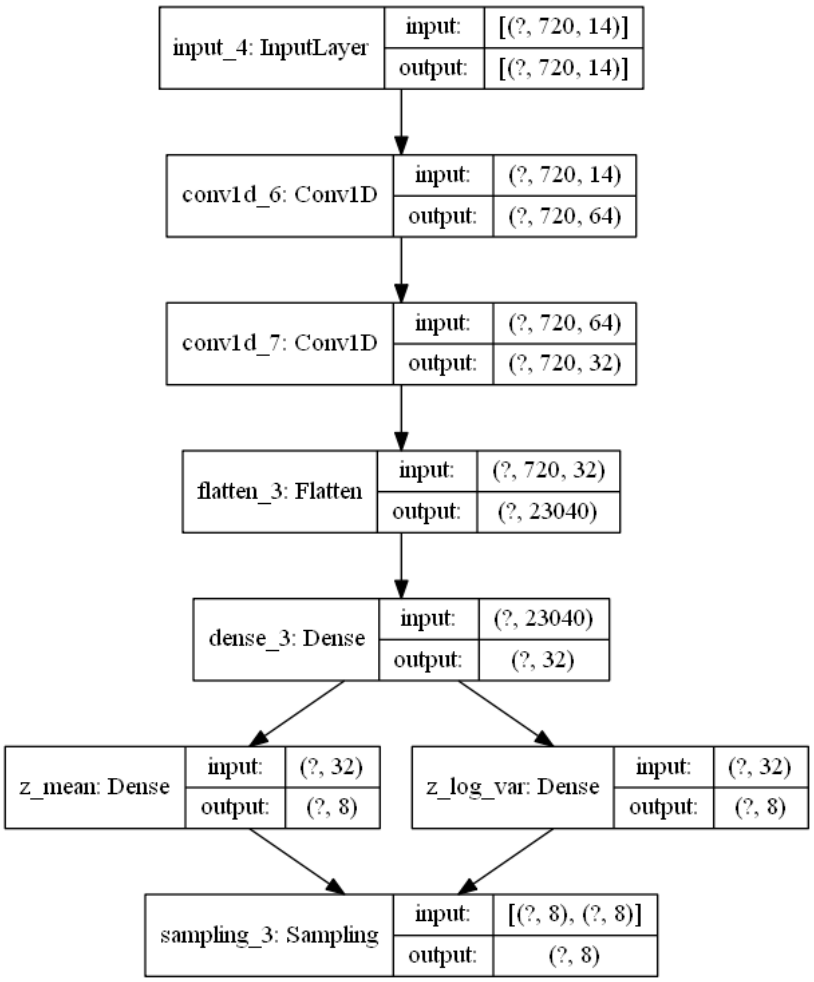

Figure 26: Structure of the encoder with the shapes of each layer (the first member of the shape in unknown until the batch size is defined)

Let's now define the decoder. First of all, an input layer with the same shape like the last layer of the encoder, which is the latent space dimension, is defined, in order to read in the encoded data. To continue, a fully-connected "Dense" layer is added, followed by a "Reshape" layer which allows us to undo what the "Flatten" layer did in the encoder, that is, go from a one single vector to a matrix of nodes, which will be connected, in turn, to several convolutional layers. Let's note here that the last convolutional layer is activated by a "SoftMax" function, instead of a ReLU. This is done because in the output, we only look for the maximum node value that the last layer has delivered, the others will be discarded, since the prediction is singular.

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(720 * 32, activation='relu')(latent_inputs)
x = layers.Reshape((720, 32))(x)
x = layers.Conv1D(32, 3, activation='relu', strides=1,
padding='same')(x)
x = layers.Conv1D(64, 3, activation='relu', strides=1,
padding='same')(x)
```

```
decoder_outputs = layers.Conv1D(14, 3, activation='softmax',
padding='same')(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name='decoder')
decoder.summary()
```

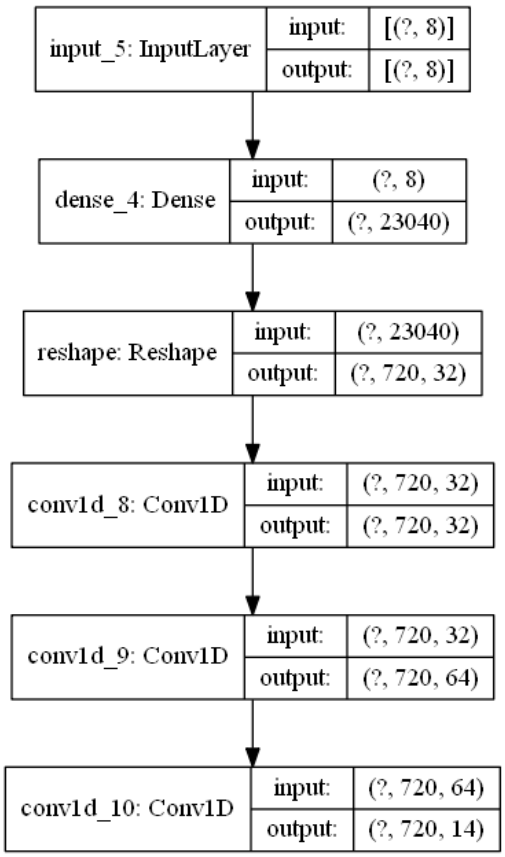

Figure 27: Structure of the decoder with the shapes of each layer (the first member of the shape is unknown until the batch size is defined)

Finally, we define a customized training step, since we need to go through the "reparameterization trick" in order to update the weights and biases of the network. This training step uses gradient descent for Back Propagation of error. The total loss consists in the sum of the reconstruction loss and the loss of the Kullback-Leibler divergence. In the reconstruction loss, we will use the "categorical cross entropy" function of "Keras", which is the function used to multi-class classification problems, like in our case, where the target values are ranged from 1 to 13 and each value indicates a different activity. The difference is calculated between the original data and the generated one from the decoder.

```
def train_step(self, data):
    data = data[0]
    with tf.GradientTape() as tape:
      z_mean, z_log_var, z = encoder(data)
      reconstruction = decoder(z)
```

```
        reconstruction_loss =
tf.reduce_mean(keras.losses.categorical_crossentropy(data,
reconstruction))
        reconstruction_loss *= 720
        kl_loss = 1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var)
        kl_loss = tf.reduce_mean(kl_loss)
        kl_loss *= -0.5
        total_loss = reconstruction_loss + kl_loss
    grads = tape.gradient(total_loss, self.trainable_weights)
    self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
    return {'loss': total_loss,
        'reconstruction_loss': reconstruction_loss,
        'kl_loss': kl_loss}
```

6.2.2.2 Data preparation

As we said, our dataset consists on sequences of activities carried out during a day by a population of individuals. Apart from the activities carried out, we also have some descriptive data about each individual, among those, the day of the week the sequence is done and the age of the person studied. In a first approach, in order to have more consistent results with patterns easier to identify, we decided to delete from the dataset all the sequences that had been done during the weekend (Saturdays and Sundays) and those which were performed by people aged more than 65 years old, which is a common age to retire from work. The reason to obviate these individuals is to obtain more similar sequences with people that goes to work during the week. With this, we have gone from a population of 11.303 individuals to 6.873. However, after the first analysis has been done, we will add these individuals again in the dataset and see how the results change.

In a more technical sense, the vectors of activity sequences had been changed into what is called one-hot encoded vectors or “dummy variables”. This has been done because our data is categorical, which means that variables contain label values instead of numeric values (Brownlee, 2017a). The number of possible values is limited, in our case from 1 to 13, and each value represents a different category, even though they are natural numbers. The problem is that the vast majority of machine learning algorithms cannot operate with categorical data directly, it needs to be converted into numbers. It’s true that in our case, categories are already converted into numbers, but since no ordinal relationships exist between the numbers, the integer encoding is not sufficient, because the network will assign more importance to some numbers than others which may result in poor performance and unexpected results (predictions halfway between categories) (Brownlee, 2017a). So what one-hot encoding does, is to create a matrix with as much columns as different categories there are. Then, each sample is a vector of 14 categories, and only the i-th term of the vector that coincides with the category number is

filled with a 1, the rest is set to 0. An example of one-hot encoding of categorical data can be seen in Figure 28.

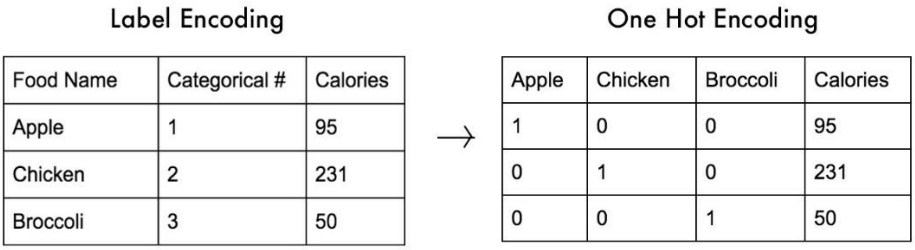

Figure 28: Example of one-hot encoding (Source: (DelSole, 2018))

6.2.2.3 Adjusting the model

Once the model has been defined, there are still some parameters which may be subject to variation and can affect the performance of the model.

The first parameter to consider is the "batch size" number that the model will take when it is being trained on the training data. The batch size defines the number of samples to use when training before updating the internal model parameters, such as the weights and biases of the different layers (Brownlee, 2018). A sample is equal to a single row of data, in our case, one sequence of 720 timesteps. The network iterates over one or more samples and makes predictions. When the batch has been finished, the predicted values are compared to the expected ones and the error is calculated to back propagate it via the stochastic gradient descent. Usually, smaller batch sizes are used for two main reasons: "they are noisy, offering a regularizing effect and lower generation error" and they "make it easier to fit one batch worth of training data in memory" (Brownlee, 2019a). Usually, a batch size of 32 or lower works well (Bengio, 2012; Masters & Luschi, 2018), however higher values like 64 or 128 may be fine for some datasets. The batch size also influences the stability of the training process and how quickly the model learns (Brownlee, 2019a).

The second parameter to consider is the number of "epochs", which consists in the total number of times that the network will train through the entire training dataset. An "epoch" is made of one or more batches (if the batch size is as big as the whole dataset, then there is only one batch). The number of epochs is usually large, letting the model run until the error is minimized enough (Brownlee, 2018). While the error (loss) keeps descending considerably, the number of epochs can be increased. The problem is that if it's higher than needed, the model can incur in

overfitting. *Overfitting* means that the model performs very well on the training data, but when new unseen data is presented, like the testing data, it is not able to deliver good results.

The third parameter to control is the latent space dimension created between the encoder and the decoder. The latent space is an abstract multi-dimensional space containing vectors of features from the encoded data, which cannot be interpreted directly but contain a lot of useful information. In some way, it is the space where the encoded data lies, waiting to be outputted to real-world data again by the decoder. The smaller the dimension of the latent space, the bigger the compression of the data. However, an excessively small latent space may induct in too much compression and we could lose some important information about the data that it won't be able to be reconstructed by the decoder later. The point is to find a right number for the latent space dimension which allows us to compress the data without losing excessive information.

A fourth parameter that is interesting to control is the percentage of training data and testing data taken from the whole dataset. A common practice is to take 75/80% of the data for training the dataset and 25/20% for testing it. In our case we will use 80% of the dataset for training and 20% for testing the results.

A fifth issue to take into account, which is not directly a parameter, is related to the population studied. Since we have sociological data, we can select our samples based on age and on the day of the week the sequence has been done. To this end, we can analyze if deleting the weekends from the dataset, for example, the results are better or worse, since there should be less variability. Also, with the ages, since retired people (aged more than 65, usually), may have more variable activity sequences than people who goes to work with a daily routine. After having tried different configurations, we have seen that with all the data (without eliminating samples based on these sociological features), the model performs better. The reason is that with more variability in the input data, we are able to explain better the behavior of the population.

To check how good the model is with different sets of the parameters, we will check two main groups of metrics: the first one related to the training data and the second one related to the testing data. Among the ones related to the training data we will check the value of the reconstruction loss. The ones related to the testing data will be related to the metrics mentioned at the beginning of section 6.2.

So, we will analyze the performance of the model based on different experiments with different values for the aforementioned parameters.

The first thing we will do is analyze the performance on the model for different configurations of the number of epochs. The batch size will be fixed to 16 and the latent space dimension to 16 too. We will deal with the whole dataset for this experiment, without excluding weekends and sequences of individuals who are older than 65 years old. The number of epochs analyzed are: 15, 30, 60, and 100. For the training part, we will analyze how the reconstruction loss performs. For the testing part, once the data has been run through the VAE, we will analyze the following metrics related to the ones mentioned in section 6.2:

- Average difference between the testing input data and the one generated by the VAE in the number of daily trips.
- Average difference between the testing input data and the one generated by the VAE in the percentage of daily travel time.
- Average difference between the testing input data and the one generated by the VAE in the number of different activities done outside from home.
- Average difference between the testing input data and the one generated by the VAE in the percentage of hours outside from home.
- Average difference between the testing input data and the one generated by the VAE in the average percentage of time spent in each activity[2].

These metrics can be understood as a sort of Mean Average Error (MAE), which computes the average of the absolute differences between the observed and the predicted data. They have been calculated the same way as the equation below:

$$MAE = \frac{1}{N}\sum_{i}^{N}|y_i - \tilde{y}_i|$$

The results can be found on the following graphs.

[2] For this metric, we will take an average of the metrics 5 to 14 mentioned in section 6.2. This has been done in order to have a simpler metric that allows us to compare different sets of parameters faster.

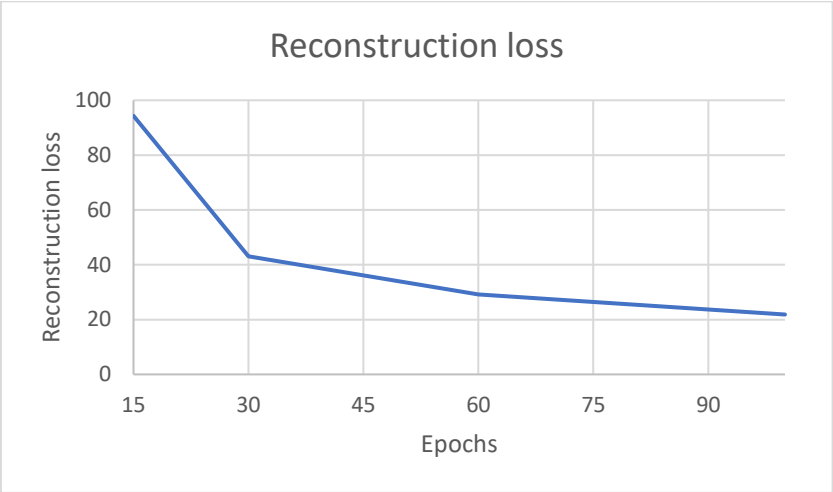

Figure 29: Evolution of the reconstruction loss for the different number of epochs

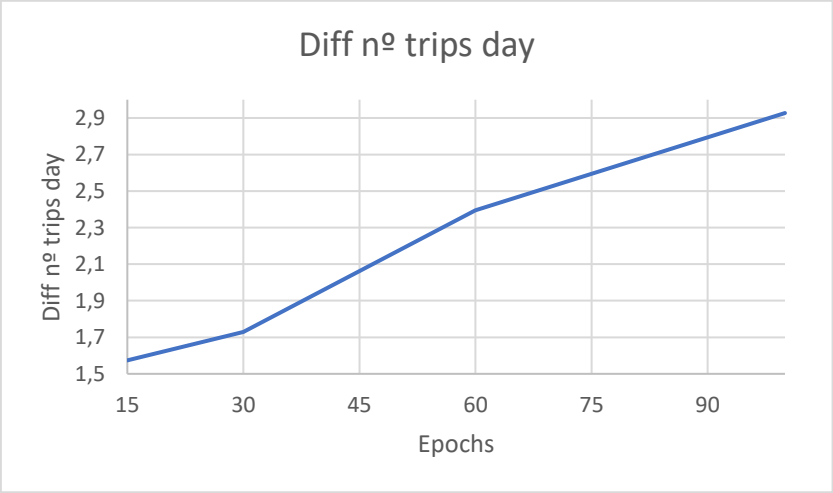

Figure 30: Evolution of the difference (testing input data vs VAE's decoded data) between the number of daily trips for the different number of epochs

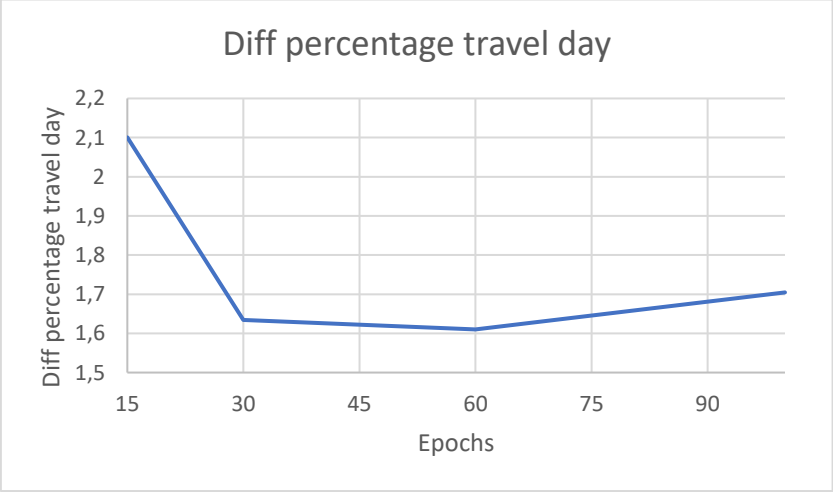

Figure 31: Evolution of the difference (testing input data vs VAE's decoded data) between the percentage of daily traveled time for the different number of epochs

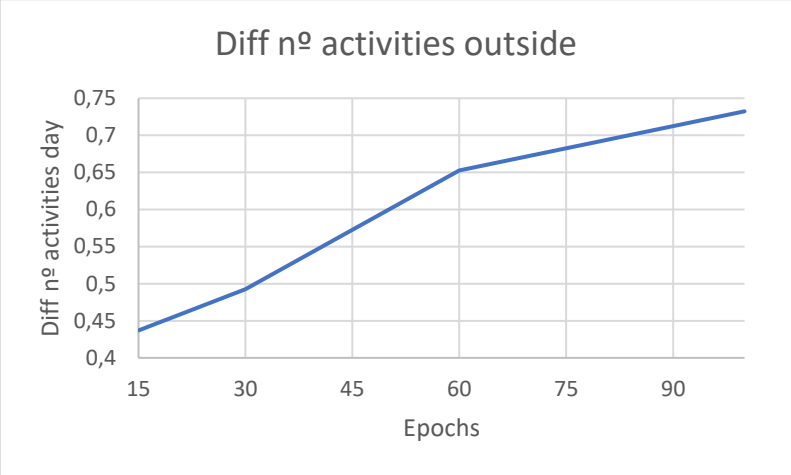

Figure 32: Evolution of the difference (testing input data vs VAE's decoded data) between the number of daily different activities done outside from home for the different number of epochs

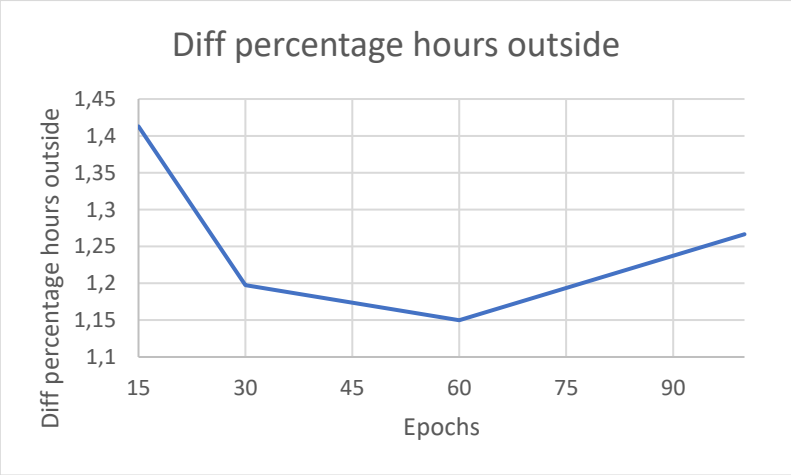

Figure 33: Evolution of the difference (testing input data vs VAE's decoded data) between the percentage of hours spent outside from home for the different number of epochs

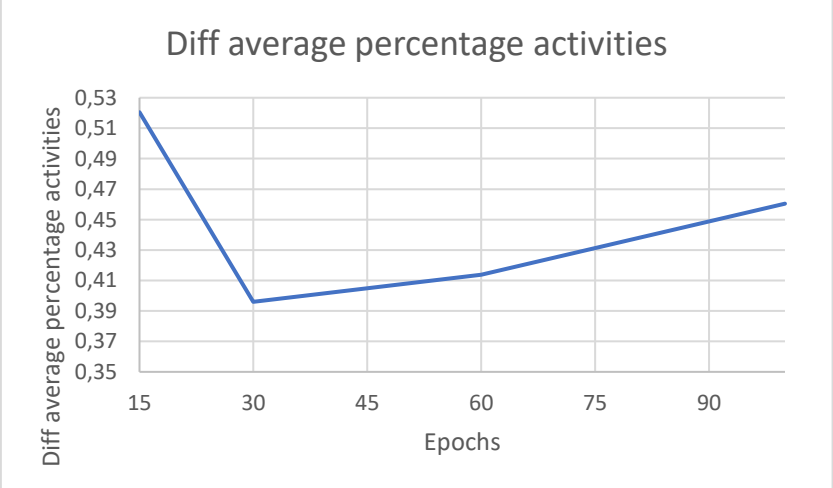

Figure 34: Evolution of the difference (testing input data vs VAE's decoded data) between the average percentage dedicated to the studied activities for the different number of epochs

As we can observe in the graphics, the higher is the number of epochs, the lower is the reconstruction loss. That makes sense, since the more we train our network, the more it will be able to learn from the data. However, as mentioned before, an excessive number of epochs can produce an *overfitting* on the training data, and when new unseen data is presented to the model, such as the reserved testing data, the model is not able to reproduce well the activity sequences. And this is, indeed, what is happening. As we can see, the model with 100 epochs almost always performs worse than with 30 or 60 epochs when it comes to reconstructing data, that means differences between the inputted and de decoded data are higher (Figures 30, 31, 32, 33, and 34). On the other hand, with 15 epochs the model has a high reconstruction loss, and effectively is less available to reconstruct the data in a close distribution to the inputted data, in other words, it is not trained enough. In Figures 31, 33 and 34 we can observe how the highest reconstructing error is with 15 epochs. So, we will discard 15 epochs for being a too low number and 100 for being too high one which causes *overfitting*. Between 30 and 60 epochs, generally 30 perform better than 60 in the testing data metrics (Figures 30, 32, and 34). Only in two metrics (difference between percentage of daily traveled time and difference between percentage of hours outside from home) 60 epochs perform better than 30, but the difference is very small (0,02 in the first case and 0,05 in the second) (Figures 31 and 33). In addition, in the difference of number of daily trips, the difference is quite high (1,73 trips in the case of 30 epochs and 2,39 trips in the case of 60 epochs). Despite both numbers are relatively high, with 60 epochs it is excessively higher. Having an average difference of 2,39 daily trips between the observed and the predicted data means that for a travelling population of 5 million people, we would have around 12 million more of trips a day, which is a high number that should be not underestimated. However, this number would not be so high in reality, since we are taking absolute differences here. For example, for a single individual, the VAE can generate less trips in a day than the original sequence for the same individual (let's say it generates 2 trips less). On the other hand, for a different individual, the VAE can generate more trips for him compared to the original one (let's say it generates 2 trips more). In practice, these two individuals would compensate between them, but since we are taking the absolute difference, we will get a difference of $\frac{|-2|+2}{2} = 2$ trips, when the real difference is 0.

For all of these reasons mentioned, we will choose 30 epochs as the definitive number, since it delivers the best performance in general terms.

The next thing to configure is the batch size and the latent space dimension. These two parameters are not so simple to tune as the number of epochs. We will take the same metrics as before. The analyzed batch sizes will be: 128, 64, 32, and 16. The analyzed latent space dimensions will be: 24, 16, 12, 8, and 4. In this case, we analyzed the sequences on weekdays (excluding weekends) and of people younger than 65 years old. As it has been explained, the neural network performs better when all the population is considered, without any exclusions. However, this discovery has been done later, and by the time of doing it, several experiments on the batch sizes and latent space dimensions had already been computed, and if we were to repeat them all it will consume a lot of time, since we would have had to train all the models again. Nevertheless, for the comparative purposes in this section, this fact doesn't suppose a problem, since the comparatives are done all with the same population. In the next graphs the results can be observed (B in the legend refers to the batch size).

Figure 35: Evolution of the reconstruction loss for the different number of batch sizes and latent space dimensions

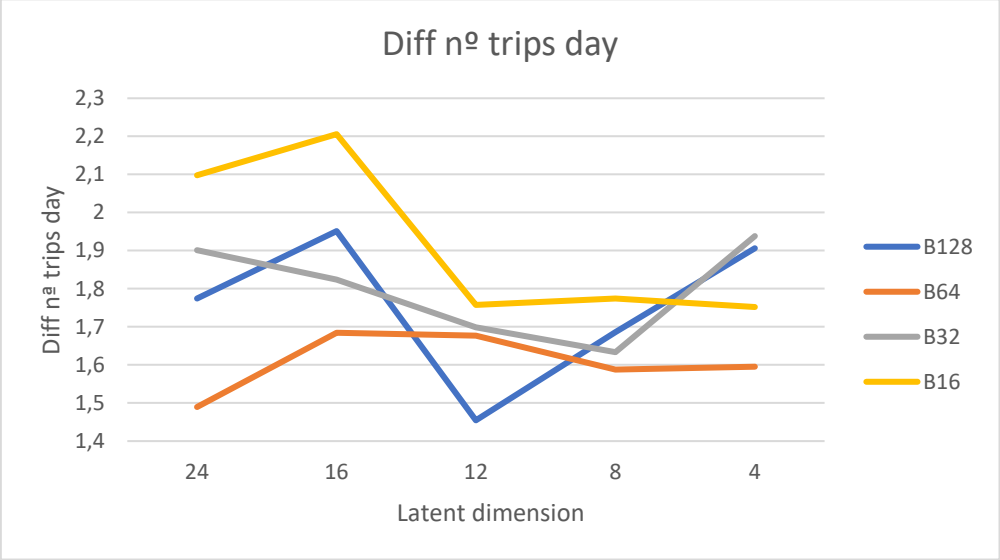

Figure 36: Evolution of the difference (testing input data vs VAE's decoded data) between the number of daily trips for the different number of batch sizes and latent space dimensions

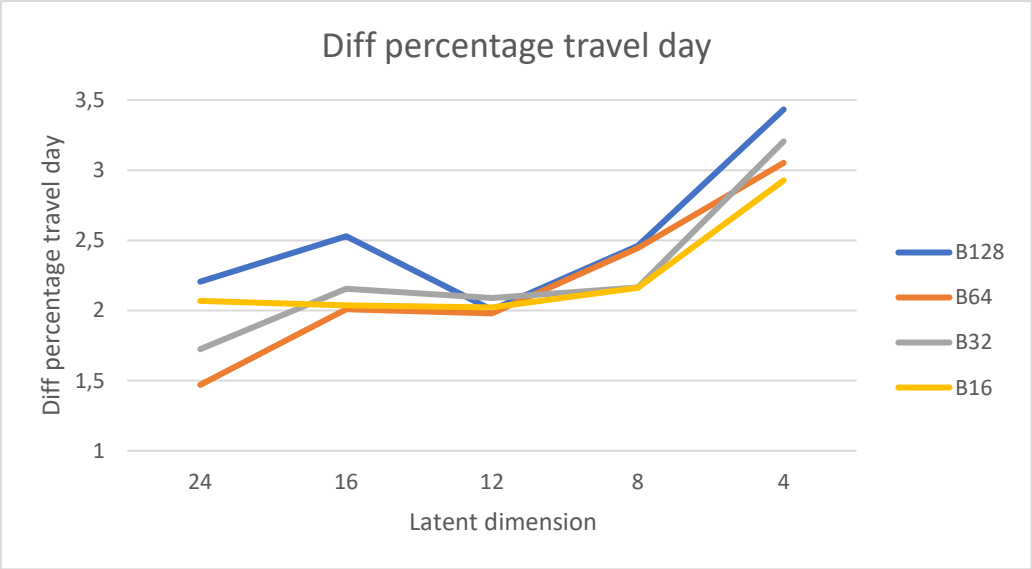

Figure 37: Evolution of the difference (testing input data vs VAE's decoded data) between the percentage of daily traveled time for the different number of batch sizes and latent space dimensions

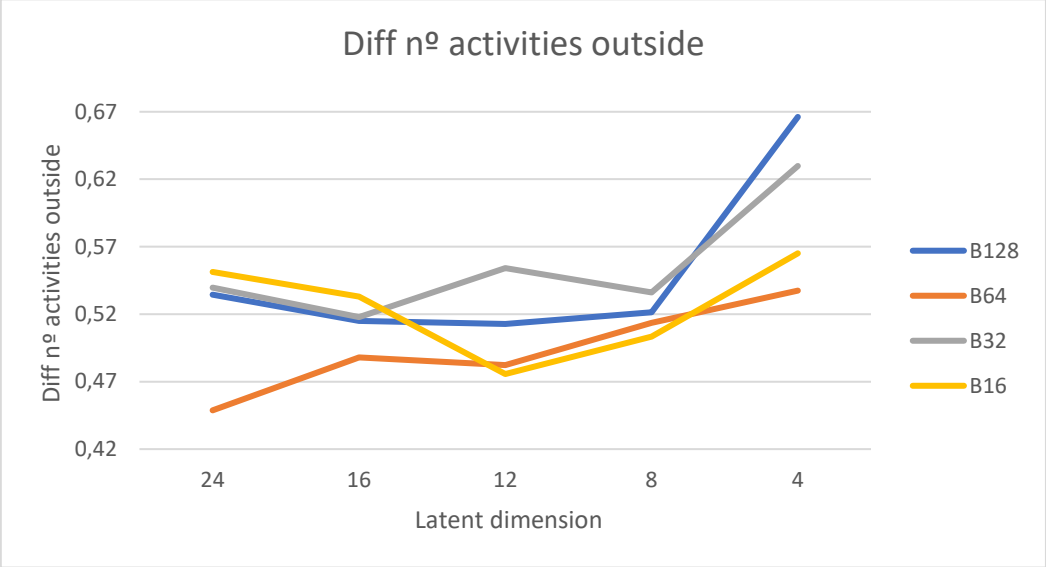

Figure 38: Evolution of the difference (testing input data vs VAE's decoded data) between the number of daily different activities done outside from home for the different number of epochs

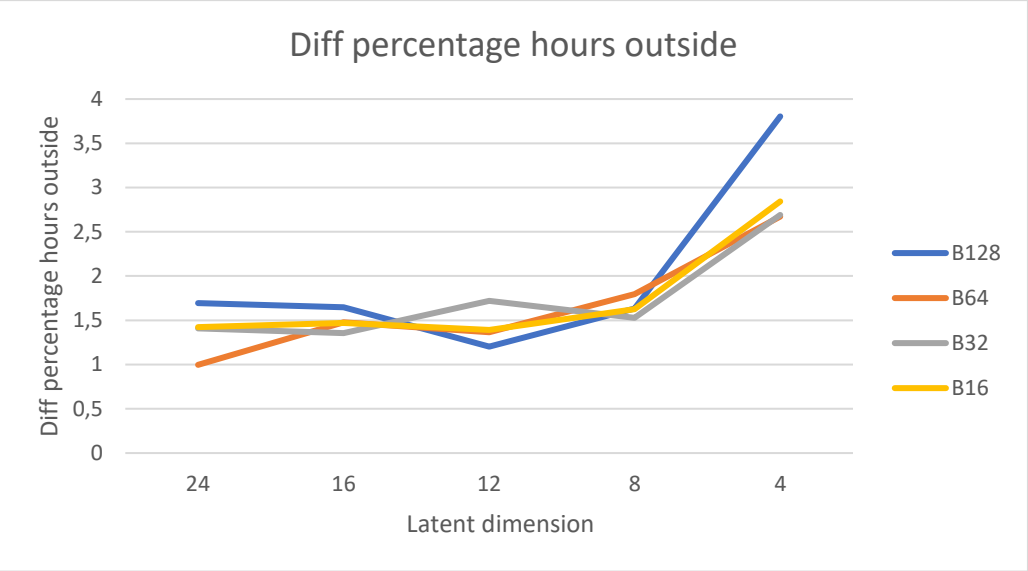

Figure 39: Evolution of the difference (testing input data vs VAE's decoded data) between the percentage of hours spent outside from home for the different number of batch sizes and latent space dimensions

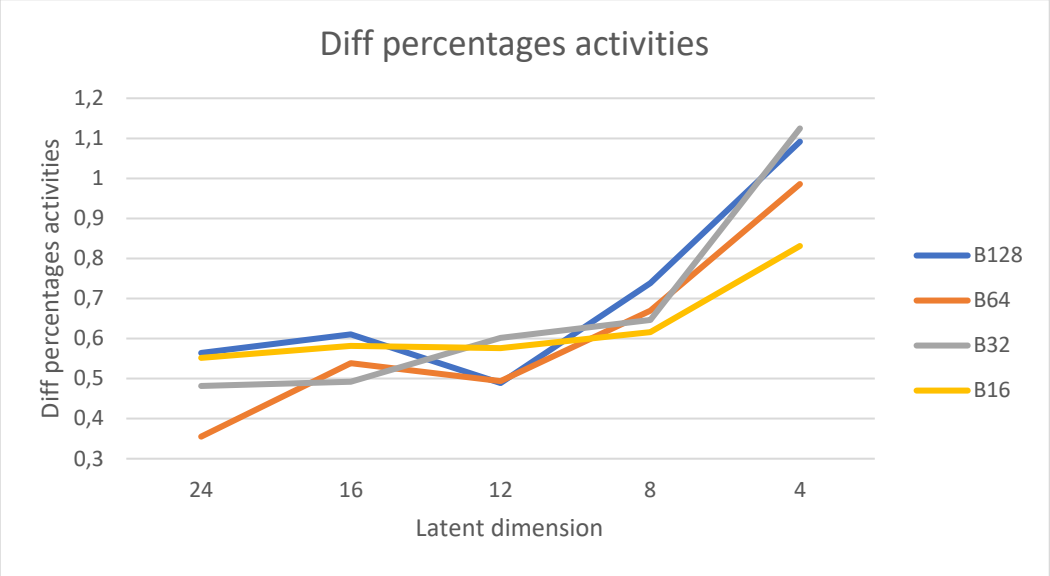

Figure 40: Evolution of the difference (testing input data vs VAE's decoded data) between the average percentage dedicated to the studied activities for the different number of batch sizes and latent space dimensions

As we can observe from Figures 35, 37, 38, 39, and 40, the model doesn't perform well with the latent space dimensions of 8 and 4. The compression done by the encoder into the latent space is too big, and some information is lost in this process. Thus, the decoder is not able to decode accurately, since the information provided to it is more incomplete than the other models with a higher latent space dimension. Normally, with a higher dimension the results should be better, since the compression done by the encoder is smaller. However, results show that there is not a significant difference between latent dimensions of 24, 16, and 12 (Figures 38, 39, and 40). In fact, the latent space dimension of 24 performs worse than the other two in the case of the difference between the number of different activities done outside from home (Figure 38). There is no big difference between a latent space dimension of 16 and 12, so both could be used for the model. Between 12 and 16 for the latent space dimension, the best batch sizes are 64 and 32. A batch size of 128 causes a high reconstruction loss, which means that the model trains worse. A batch size of 16 results in a lower reconstruction loss but in higher differences between the testing data and the decoded data. Between a batch size of 32 and 64, usually 64 performs better than 32 (Figures 42, 43, and 44), especially when it's with a latent space dimension of 12 (Figures 41, 45, and 46).

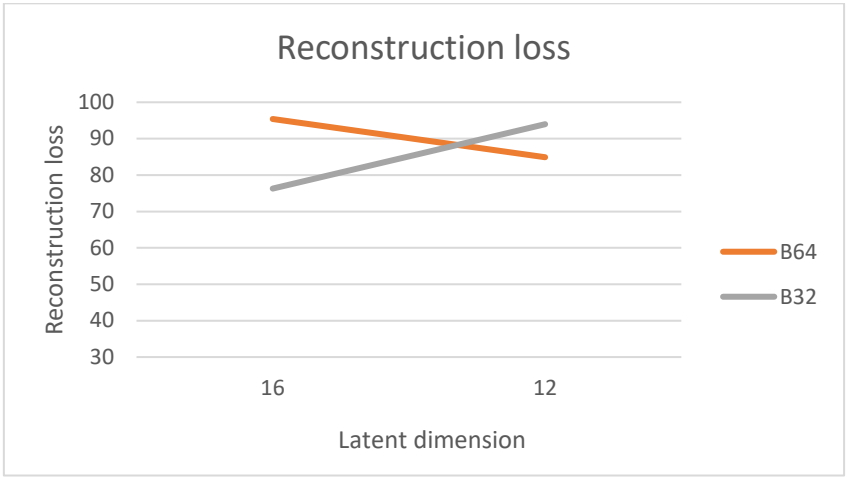

Figure 41: Reconstruction loss for the batch sizes (32 and 64) and latent space dimensions (12 and 16)

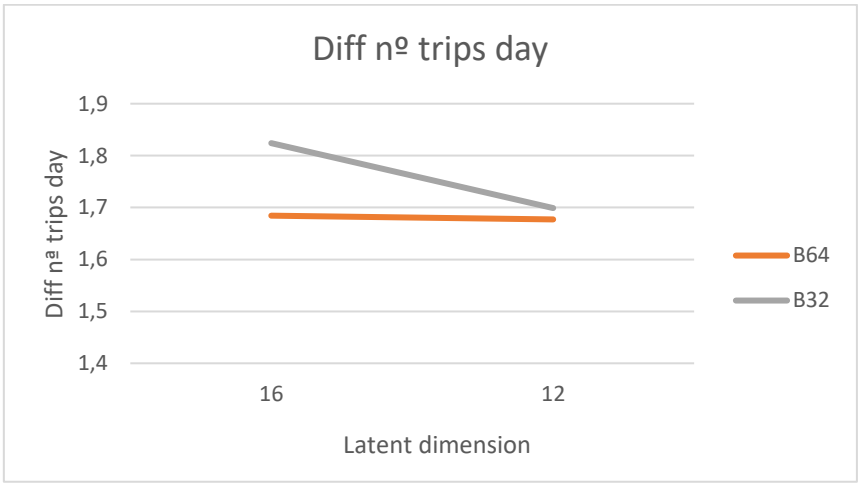

Figure 42: Difference (testing input data vs VAE's decoded data) between the number of daily trips for the batch sizes (32 and 64) and latent space dimensions (12 and 16)

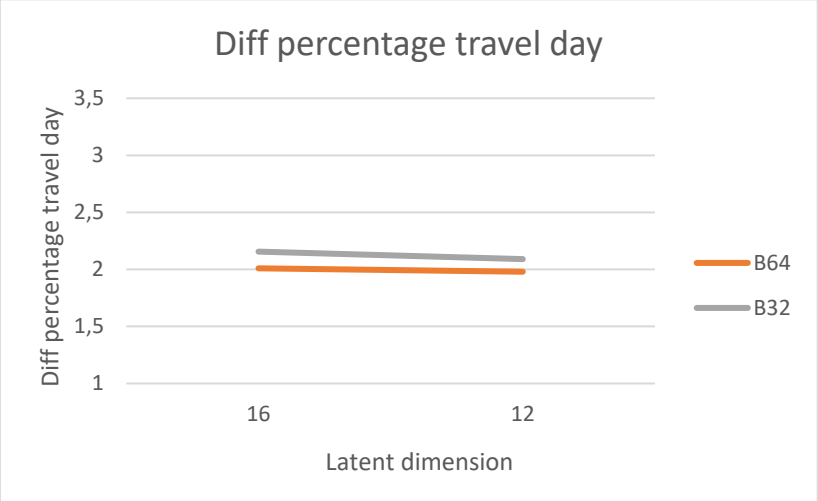

Figure 43: Difference (testing input data vs VAE's decoded data) between the percentage of daily traveled time for the batch sizes (32 and 64) and latent space dimensions (12 and 16)

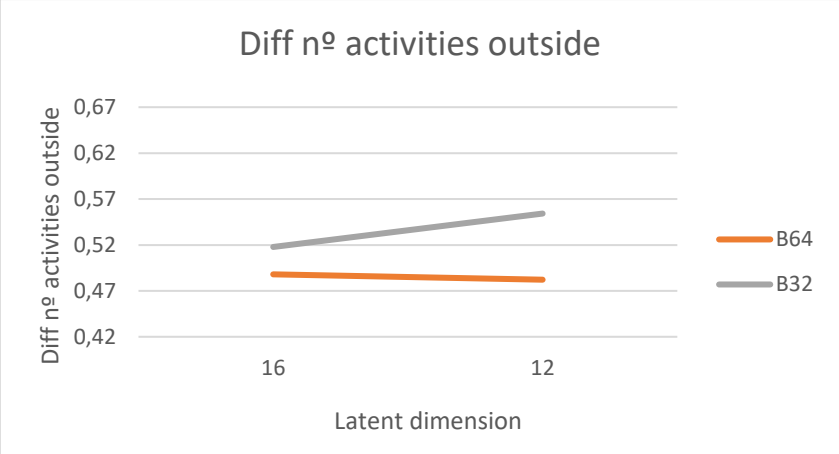

Figure 44: Difference (testing input data vs VAE's decoded data) between the number of daily activities done outside from home for the batch sizes (32 and 64) and latent space dimensions (12 and 16)

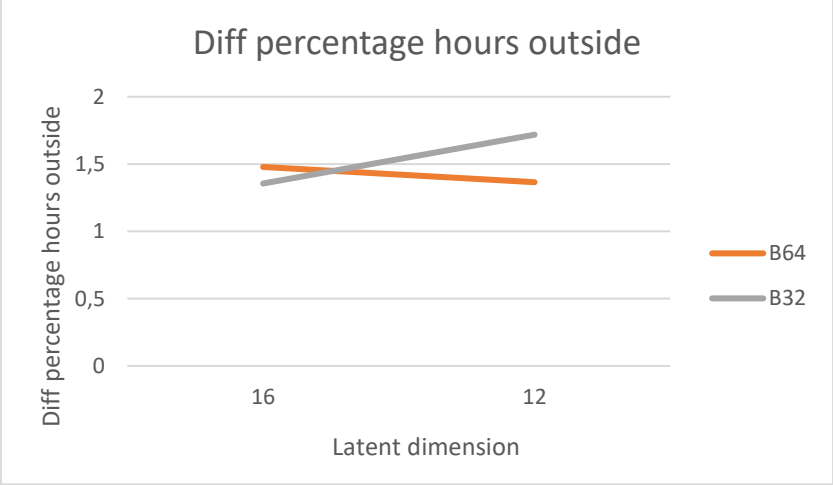

Figure 45: Difference (testing input data vs VAE's decoded data) between the percentage of hours spend outside from home for the batch sizes (32 and 64) and latent space dimensions (12 and 16)

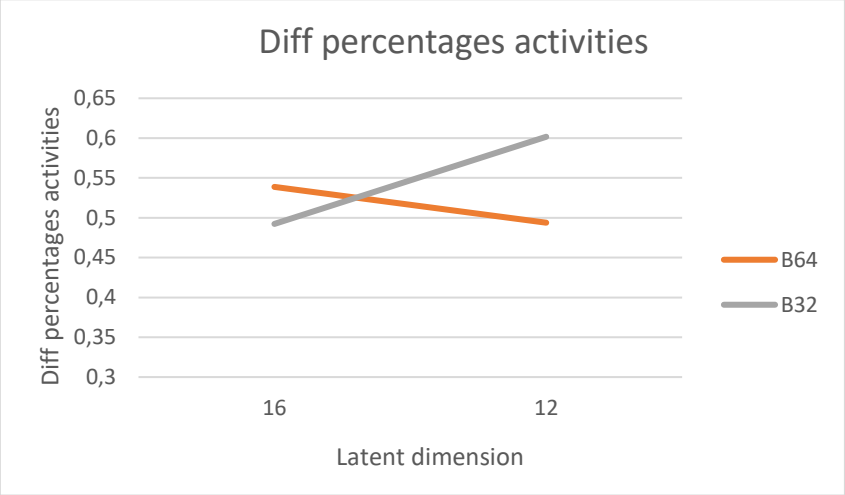

Figure 46: Difference (testing input data vs VAE's decoded data) between the average percentage dedicated to the studied activities for the batch sizes (32 and 64) and latent space dimensions (12 and 16)

So finally, the parameters chosen will be 30 epochs, a latent space dimension of 12 and a batch size of 64.

6.2.2.4 Final model results

Let's now analyze the results for the final chosen model (30 epochs, latent space dimension of 12, and batch size of 64). To do so, we will analyze the metrics mentioned in section 6.2.

Metric	Value (VAE)	Value (BELDAM)
Number of trips carried out by an individual	3,885	3,117
Number of activities carried out outside from home by an individual	1,513	1,486
Percentage of hours spent traveling by an individual	5,45 %	5,379 %
Percentage of hours that a person spends outside from home	28,53 %	28,7 %
Percentage of hours that a person spends at work	9,889 %	9,921 %
Percentage of hours that a person spends at home	71,47 %	71,3 %
Percentage of hours that a person spends at work trips	0,656 %	0,653 %
Percentage of hours that a person spends following courses	4,217 %	4,287 %
Percentage of hours that a person spends having a meal outside	0,4337 %	0,4399 %
Percentage of hours that a person spends shopping	1,781 %	1,741 %
Percentage of hours that a person spends taking services	0,4745 %	0,5103 %

Metric	Value (VAE)	Value (BELDAM)
Percentage of hours that a person spends visiting family or friends	2,344 %	2,219 %
Percentage of hours that a person spends walking or making a tour	0,6045 %	0,651 %
Percentage of hours that a person spends in leisure, culture or sports	1,467 %	1,488 %

Table 3: Value of the proposed metrics for the VAE model

As we can derive from Table 3, the model based in a Variational Autoencoder is much better than the one based in a Frequency Analysis at the time of generating new activity sequences. Not only the differences of the metrics based in percentages are lower, but also the ones based in the number of daily trips and in the number of different activities done outside from home. This means that at the disaggregate level, the model is more able to grasp the features of the data, and moreover, the alternance between an activity of any kind and a trip in between. Even though this alternance is not always present, in the great majority of sequences, it is.

The negative point that we could consider about this VAE model, is that it generates 0,768 more daily trips in average. Even though this difference is not very high in absolute numbers, for a big population, like a whole country, this supposes a lot more of daily trips that need to be taken into account. For a population of 7 million people, for example, this would suppose 5,376 million trips more each day.

6.2.3 Comparison between Frequency Analysis and VAE

A total number of 35 average trips done in the FA in a day versus 3,2 average trips done by the original data. And almost 11 different activities done daily outside of home in the FA results versus 1,5 different activities done by the original population data. This is happening because the model is not able to grasp the main features of the data. One of these features is that there always exists an alternation between an activity of any kind and a trip, in other words, an alternation between a number from 1 to 12 and a 13. In Figure 47 below, we can see a graph of a daily activities sequence generated by the FA model, and we can observe how there is no clear human pattern, since there are a lot of different activities in a short period of time. Also, the alternation of numbers with 13 is not correctly done, as we can observe clearly from 13:10 h to 15:20 h, for example.

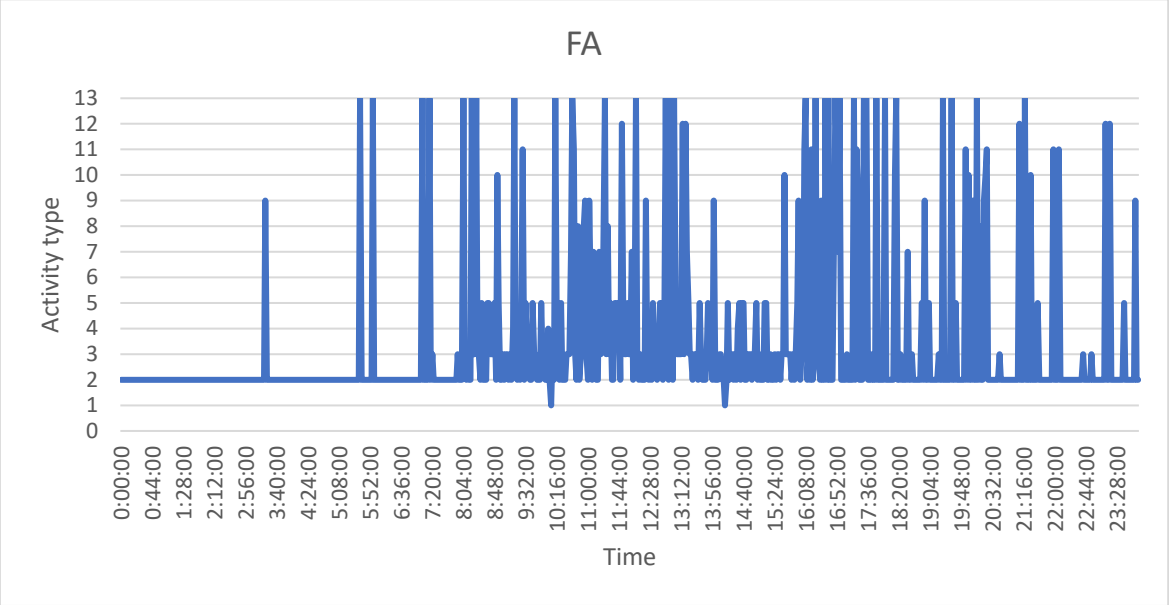

Figure 47: Activity sequence of a full day generated by the FA model

On the other hand, the VAE is capable of learning this feature, and usually intercalates a 13 between a pair of activities. In Figure 48, we can see an example of a daily sequence generated by the VAE. The person is at home from 00:00 h until approximately 08:00 h. Then he does a trip (13) to work (3), where he/she stays until 17:20 h. Then he/she takes another trip to go to visit a relative or a friend (9). Then he takes another trip to go eating outside (6), and finally another trip to return home at 19:20 h approximately, where he/she stays until the end of the day. In addition of correctly adding trips between activities, also the timing of each activity makes sense. Leaving home to work at 08:00 h is pretty common, and also leave work around 17:00 h. It is also common to go visit a relative or a friend after work and after go pick up some food for dinner. The hour of returning at home is also quite standard.

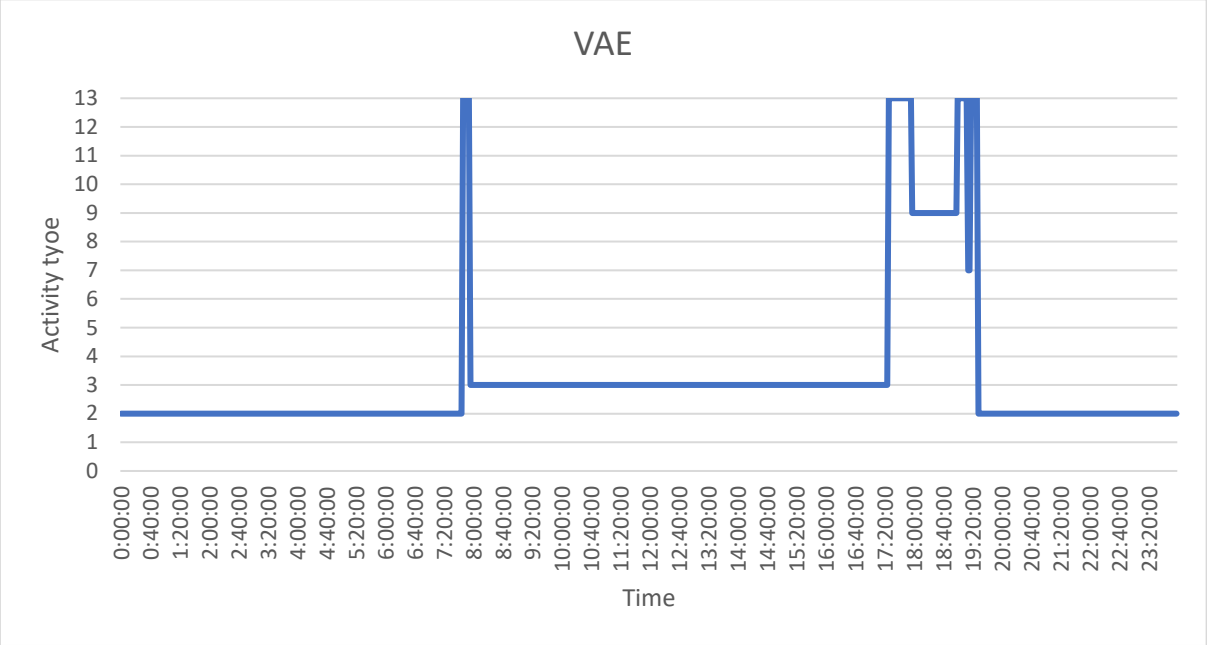

Figure 48: Activity sequence of a full day generated by the VAE

For this reason, we are very proud about the performance of the Variational Autoencoder when dealing with the BELDAM dataset. The model is able to capture the percentage of time dedicated to each activity every day and also to sequence these activities in a proper way, alternating activities with trips and positioning each activity in a time slot that makes sense for them. Clearly, the VAE outperforms the FA model at the time of generating activity schedules.

6.3 Predicting activity sequences

6.3.1 Description of the model

As it has been said in section 5.2, the LSTM cells are very used in problems where we want to predict a sequence based on a previous sequence of data (Seq2Seq), since they are capable of keeping information from a considerable large amount of time, computationally speaking. Their ability of letting important information in and eliminating useless information makes them a perfect candidate for this kind of problems. As pointed out in section 5.3, we will try to create an encoder-decoder LSTM-based neural network in order to predict activity sequences based on the information of previous done activities in the same day. The idea is to predict the next Y timesteps based on the information provided in the previous X timesteps ($X + Y < 720$). The model is also developed in a Python language environment, with the “Tensorflow” and “Keras” frameworks mentioned in section 6.2.2.1.

The model developed in this section includes two recurrent neural networks, one to encode the input sequence, the encoder, and one to decode the encoded sequence into the output sequence, the decoder. Both are based on a LSTM neural network structure. To begin with, we create a function to define the encoder-decoder structure and how it has to work, both in the training part and in the inference one.

The function takes three arguments, which are:

- ***n_input***: the cardinality of the input sequence, that is, the number of features for each timestep. In our case the value is 14, which comes from 13 different activities plus one (number 0).
- ***n_output***: the cardinality of the output sequence, that is, the number of features for each timestep. In our case the value is 14, which comes from 13 different activities plus one (number 0).
- ***n_units***: the number of cells to create in the encoder and decoder models. In our case will be 128.

The function creates and returns three models, which are:

- ***train***: model consisting in the encoder-decoder LSTM-bases structure that can be trained given source, target, and shifted target sequences.
- ***inference_encoder***: encoder model used when making a prediction for a new sequence.
- ***inference_decoder***: decoder model used when making a prediction for a new sequence.

The training in the model happens by giving source and target sequences, where the model takes both the source and a shifted version of the target sequence as input and predicts the whole target sequence (Brownlee, 2017c). For example, one source sequence may be [2, 2, 13, 13, 3, 3] and the target sequence [3, 3, 13, 13, 6]. The inputs and outputs of the model during the training phase would be:

- Input 1: [2, 2, 13, 13, 3, 3]
- Input 2: [0, 3, 3, 13, 13]
- Output: [3, 3, 13, 13, 6]

The model is called recursively when generating new sequences from source sequences. The source sequence is encoded and the target sequence is created one element at a time, using a “start of sequence” character such as “0” to start the process.

When we are predicting sequences, the ***inference_encoder*** model is used to encode the source sequence, a process that generates a return states that are used to initialize the ***inference_decoder*** model. From there, the ***inference_decoder*** is used to predict activities step by step.

```
# returns train, inference_encoder and inference_decoder models
def define_models(n_input, n_output, n_units):
    #n_input = number of features of input sequence
    #n_output = number of features of output sequence
    #n_units is the dimension of the latent space

    # define training encoder
    encoder_inputs = Input(shape=(None, n_input))
    encoder = LSTM (n_units, return_state = True)
    encoder_outputs, state_h, state_c = encoder(encoder_inputs)
    encoder_states = [state_h, state_c]
    #define training decoder
    decoder_inputs = Input(shape=(None, n_output))
    decoder_lstm = LSTM(n_units, return_sequences = True,
return_state=True)
    decoder_outputs, _, _= decoder_lstm(decoder_inputs,
initial_state=encoder_states)
    decoder_dense = Dense(n_output, activation='softmax')
    decoder_outputs = decoder_dense(decoder_outputs)
    model= Model([encoder_inputs, decoder_inputs], decoder_outputs)
    #define inference encoder
    encoder_model = Model(encoder_inputs,encoder_states)
    #define inference decoder
    decoder_state_input_h = Input(shape=(n_units,))
    decoder_state_input_c = Input(shape=(n_units,))
    decoder_states_inputs = [decoder_state_input_h,
decoder_state_input_c]
    decoder_outputs, state_h, state_c = decoder_lstm(decoder_inputs,
initial_state=decoder_states_inputs)
    decoder_states = [state_h, state_c]
    decoder_outputs = decoder_dense(decoder_outputs)
    decoder_model = Model([decoder_inputs] + decoder_states_inputs,
[decoder_outputs] + decoder_states)
    #return all models
    return model, encoder_model, decoder_model
```

After having defined the models, the function ***predict_sequence()*** can be used to generate a target sequence given a source sequence. The functions take five arguments, which are:

- ***infenc***: encoder model used for inference when making a new prediction.
- ***infdec***: decoder model used for inference when making a new prediction.
- ***source***: source sequence from which we will make the prediction.
- ***n_steps***: number of timesteps to predict in the target sequence.
- ***cardinality***: cardinality of the target sequence, the number of features, which as we said before, is 14.

The function returns a vector containing the predicted target sequence.

```
def predict_sequence(infenc, infdec, source, n_steps, cardinality):
    #encode
    state = infenc.predict(source)
    #start of sequence input
    target_seq = np.array([0.0 for _ in
range(cardinality)]).reshape(1, 1, cardinality)
    #collect predictions
    output = list()
    for t in range(n_steps):
        #predict next char
        yhat, h, c = infdec.predict([target_seq] + state)
        #store prediction
        output.append(yhat[0,0,:])
        #update state
        state = [h,c]
        #update target sequence
        target_seq = yhat
    return np.array(output)
```

To train the model, we will use the “RMSPROP” optimizer from “Keras” and the loss function will be calculated as the “categorical cross entropy” between the input and the output data.

6.3.2 Data preparation

As it has been stated, data for this kind of models needs to be prepared in a special way. We don’t just need input and output sequences, but also a shifted version of the output sequence. To do this, we defined the function ***split_sequence()***, which splits a univariate sequence into the three samples. The function takes three inputs, which are:

- ***sequence***: the sequence from where to get split samples.
- ***n_steps_in***: the number of timesteps for the source sequence (the X from previous section).
- ***n_steps_out***: the number of timesteps to predict in the target sequence (the Y from previous section).

It returns three sequences, the source one, the target one, and the shifted version of the target sequence. The idea is to get smaller slices from the bigger sequence of 720 timesteps (a whole day).

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
    #X is input sequence, y is output sequence, y_1 is output sequence
one timestep forward
    #the last n_steps_in + n_steps_out are omitted because they would
be out of range (problem?)
```

```
X, y, y_1 = list(), list(), list()
i = 0
end_ix = i + n_steps_in
# gather input and output parts of the pattern
seq_x = sequence[i:end_ix]
seq_y_1 = sequence [end_ix:end_ix+n_steps_out]
seq_y = np.concatenate((array([[0]]),seq_y_1[:-1]))
if len(seq_y) == n_steps_out and len(seq_y_1) == n_steps_out:
    X.append(seq_x)
    y.append(seq_y)
    y_1.append(seq_y_1)
return array(X), array(y), array(y_1)
```

In addition, after having the sequences prepared, we need to convert them to "dummy variables" or one-hot encoded vectors, the same way as we did in section 6.2.2.2. Also, we have gone from timesteps of 2 minutes to timesteps of 6 minutes, by eliminating the ones in between. The reason to do this is to have shorter vectors which will be easier to predict. The good point is that information is not lost in the process, since there are no activities nor trips that last for only six minutes in our dataset. Furthermore, in order to not having sequences with long periods being at home, we have cut the sequences from 10:00 h to 21:00 h. We will predict what the person will do from 17:00 h to 21:00 h based on what he/she has done from 10:00 h to 17:00 h. We have done that after realizing that long periods at home affected the training of the model, with a tendency of delivering results consisting on being at home all the time. Finally, activity sequences from weekends and from people older than 65 years old have been eliminated in order to have less variability when training.

6.3.3 Results

When the model is trained, we observe considerable good results. At the end of the last epoch, the value for the accuracy is 0,9630 (96,30 %) for the trained data, which consists of 4800 samples. The value for the validation accuracy for the validation data, which consists of 1200 samples, reaches a value of 0,9650 (96,50 %) at the end of the last epoch. So, at a first glance, it seems that the model has been correctly defined and it trains properly. However, in these LSTMs models, inference is the tricky part.

And effectively, when we start making predictions, results are not so satisfying. After analyzing the testing data, consisting on 873 samples, we can observe three main drawbacks.

The first drawback is that the model is not able to detect small changes in the activity sequence. In Figure 49 we can observe an original sequence from an individual from 10:00 h to 21:00 h. We can see that he/she is at work (3) until 16:12 h, then he makes a trip (13) to go pick/drop

someone (1), and then he does another trip to return back home at 17:36 approximately (2). Probably, on the same way back home, he picked up or dropped someone, which is a common practice, among co-workers, for example.

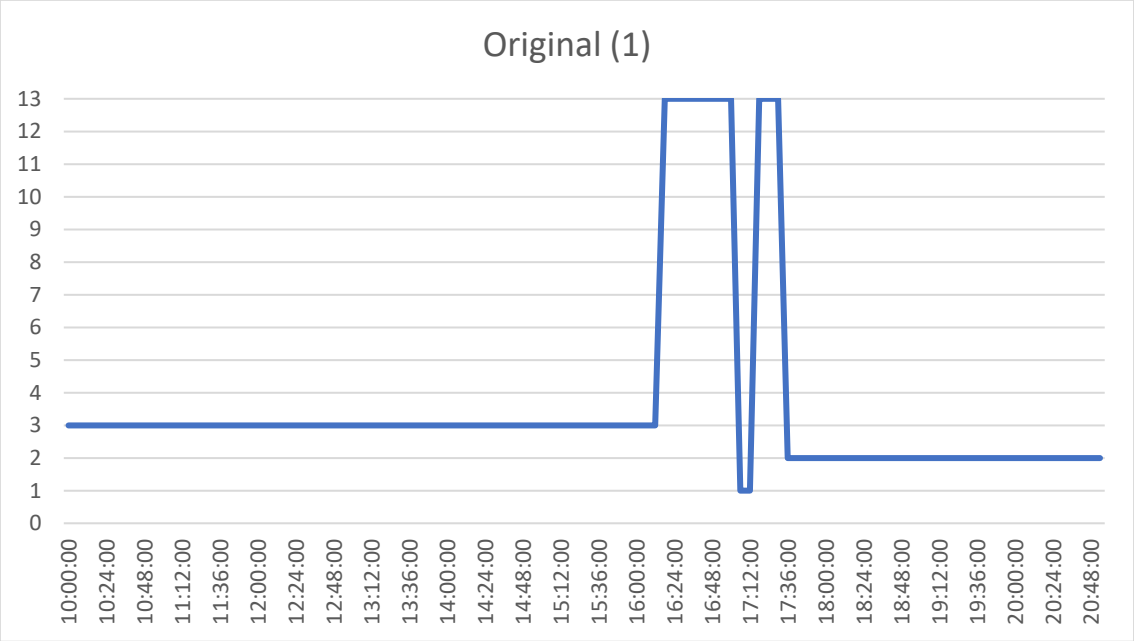

Figure 49: Original activity sequence of individual 1 from 10:00 h to 21:00 h

Now, when we observe the predicted sequence from 17:00 h to 21:00 h for this person, we can see that the picking up/dropping activity doesn't appear (Figure 50).

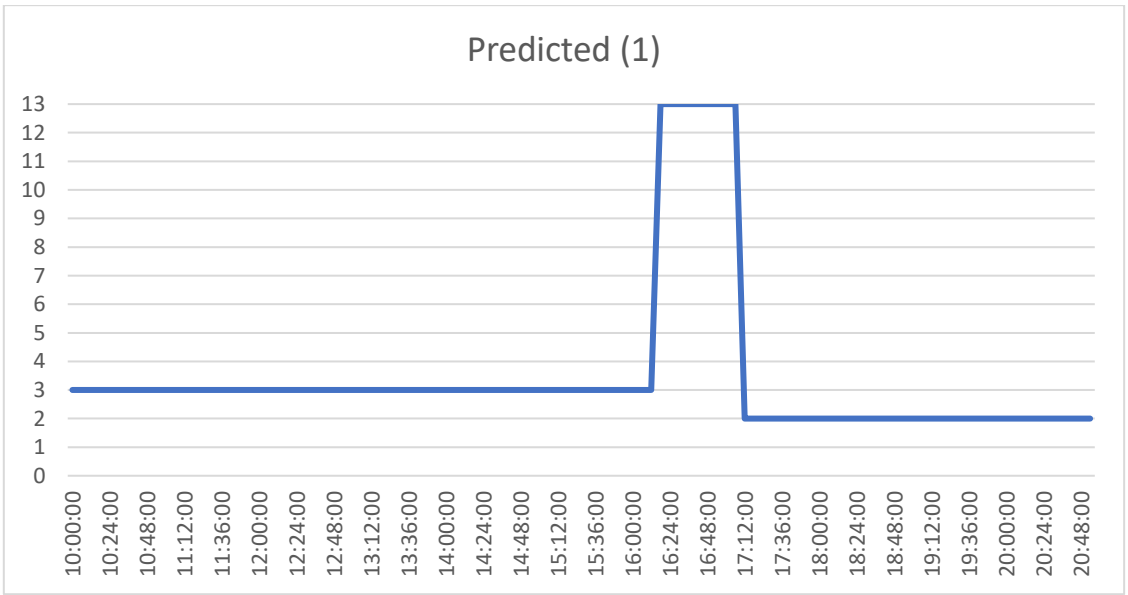

Figure 50: Predicted activity sequence of individual 1 from 10:00 h to 21:00 h

From these observation and similar ones, we deduce that the model is not able to detect these small changes in the activity sequence, in other words, it is only able to work with simple sequences, with not a lot of changes in between. In the example below, we can see how the model predicts correctly the sequence. It is a very simple sequence from a student who is following a course (5) until 15:42 h, then he/she makes a trip (13) for 24 minutes to return back home (2) at 16:18 h.

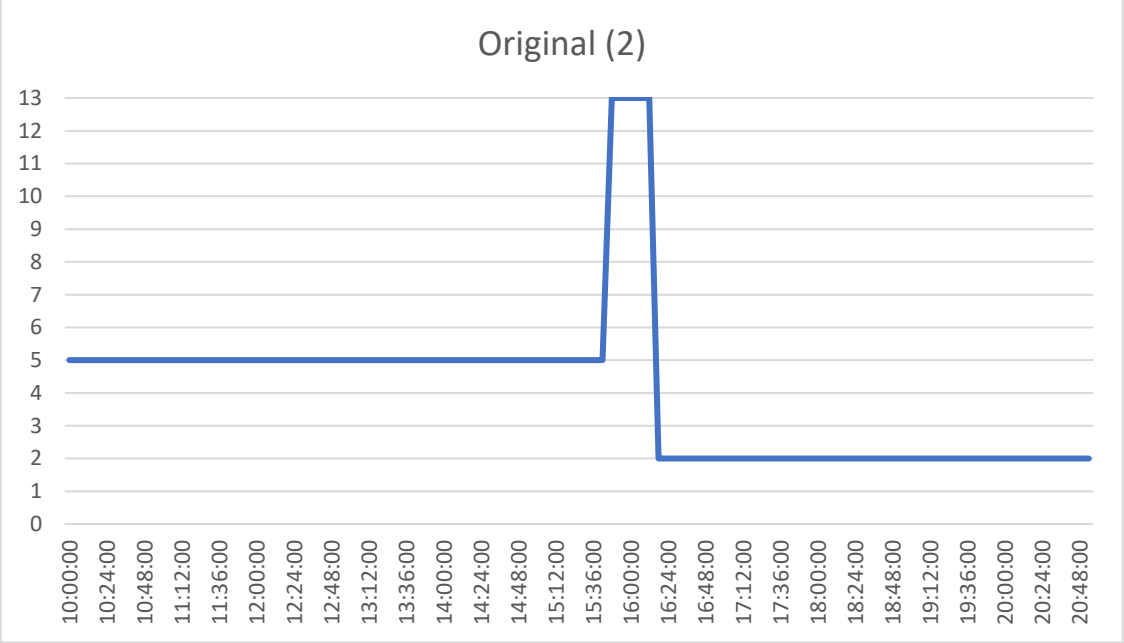

Figure 51: Original activity sequence of individual 2 from 10:00 h to 21:00 h

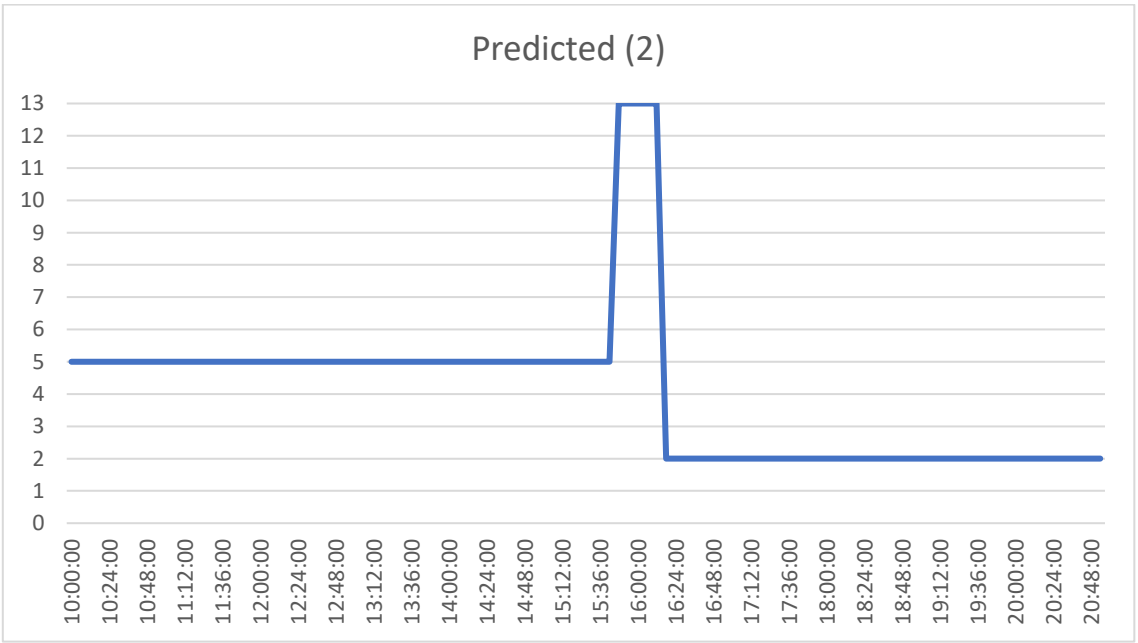

Figure 52: Predicted activity sequence of individual 2 from 10:00 h to 21:00 h

The second drawback that the model has when making predictions is that it shows a tendency to end sequences with a trip (13) to return back home (2). This is probably due to an *overfitting* on the data because it has seen that it is very common to end the day like this. We have tried to reduce the number of epochs and changing the batch size but this problem still persists. It could be related to the nature of the data and the way the model is defined. In the two graphs below we can see how the model directly cuts the sequence that there is supposed to be from 17:00 h to 21:00 h and inputs a sequence of a trip (13) followed by being at home (2). In the original sequence (Figure 53), the individual takes a work-related trip (4) for three hours approximately, then he/she goes back to work (3) for a while, and then takes another trip to return back home (2). But in the predicted sequence (Figure 54), after being at work (4), the work-related trip only lasts for 1 hour (10 timesteps of 6 minutes), and then he/she makes a trip (13) of 1 hour and 12 minutes to go back home (2). And this happens for a considerably high number of samples.

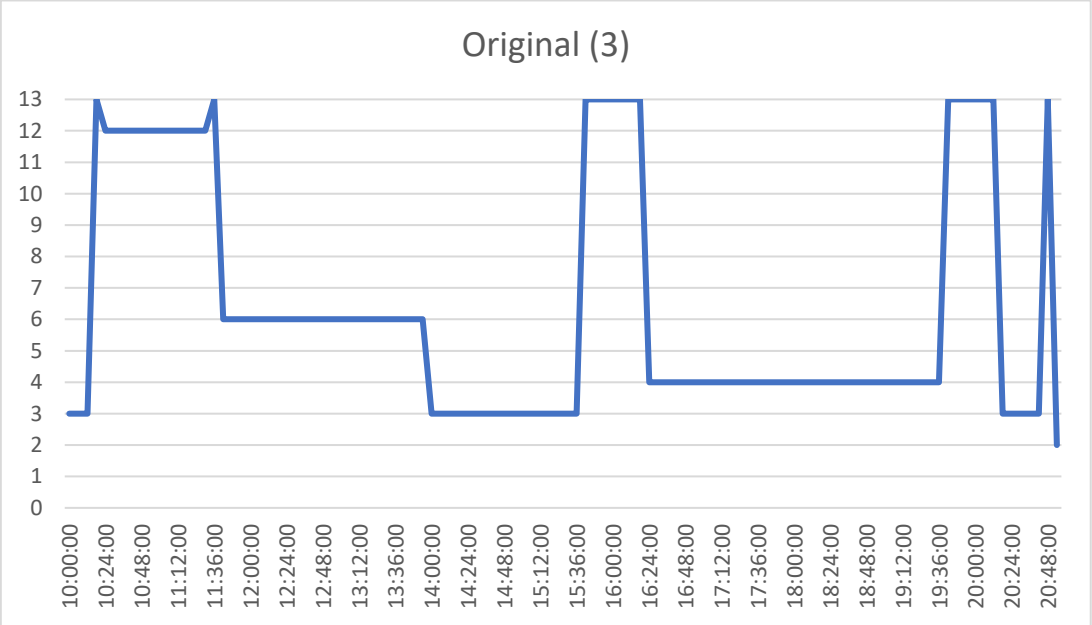

Figure 53: Original activity sequence of individual 3 from 10:00 h to 21:00 h

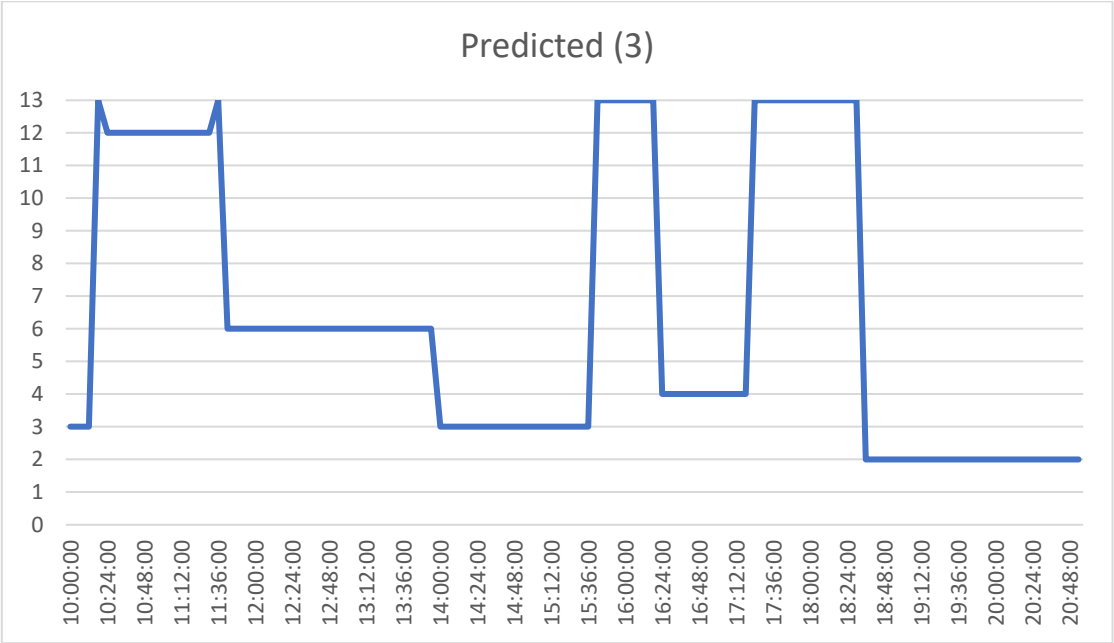

Figure 54: Predicted activity sequence of individual 3 from 10:00 h to 21:00 h

The third drawback observed, is that the model is not capable of understanding human behavior, as the VAE is able to do. For example, if someone is at work, at some point, he/she has to make a trip to leave the office, normally at home. But in our model, if there is a relatively long sequence of being at work and it continues further than the border point (17:00 h), the predicted sequence stays at the same point all the time until the end (21:00 h), without adding any activity nor a trip. We can see an example below.

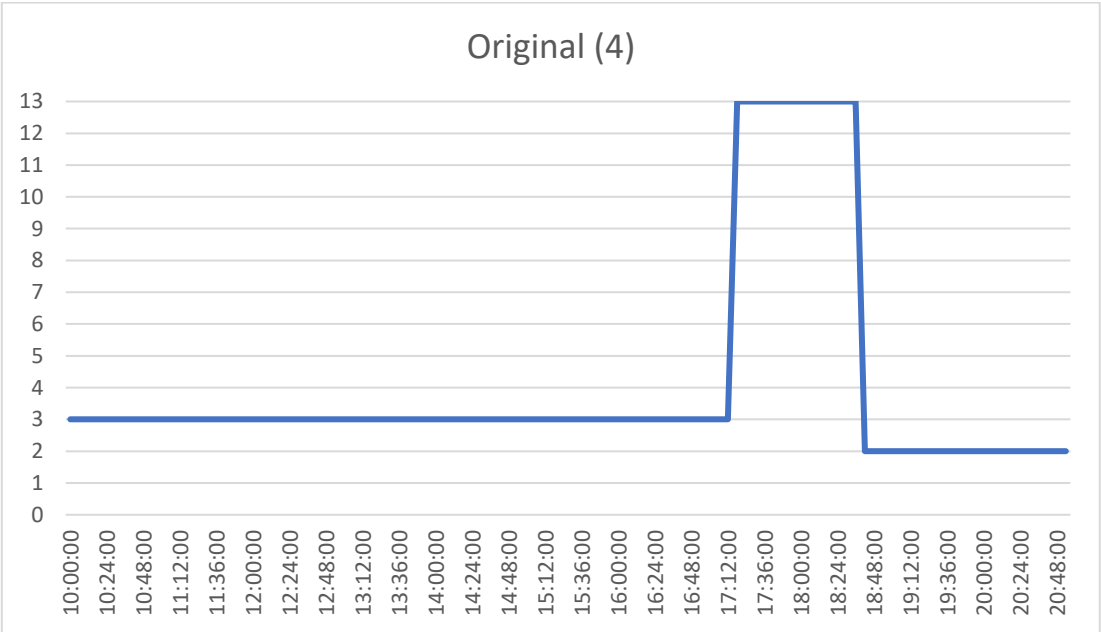

Figure 55: Original activity sequence of individual 4 from 10:00 h to 21:00 h

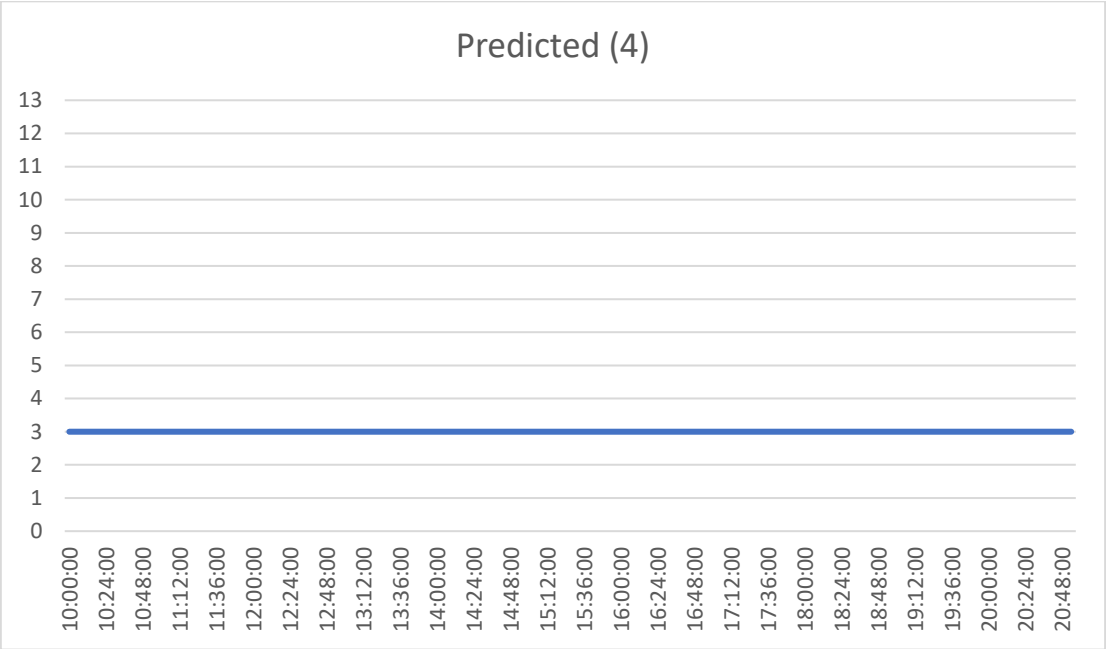

Figure 56: Predicted activity sequence of individual 4 from 10:00 h to 21:00 h

For these three reasons, we conclude that the model is not ready enough to be applied in real-life applications. However, with further training, investigation, and preparation of the data, it could be possible to get acceptable results. All in all, for now, we can only consider these predictions as early results, that show us a path of predicting activity sequences based on the activities realized previously during the day with the Neural Networks approach. Sadly, there is no more time to keep investigating with the model at the due date of this project. Also, one of the promotors got sick in the last stage of this project and he was not able to follow the work, and this considerably affected this part dedicated to the LSTM architecture.

7 Final comments and conclusions

This final chapter looks back on the main achievements of this master thesis. The summary of the proposed objectives and the obtained results are presented. Finally, some recommendations for further research are formulated.

7.1 Conclusions

This master thesis deals with the human activity sequence generation and forecasting for transport purposes. As it has been shown in the state of the art (sections 2.2 and 6), research in this line has been very active in the last 40 years. This is because traffic prediction and human behavior play a key role in mitigating some traffic and transportation problems. In particular, this project is focused on activity sequences generation and prediction using activity sequences derived from trip diaries from the BELDAM report about the Belgian population. The objectives presented in section 1 are going to be revised and some conclusions about them are going to be stated.

Thus, the main objective of this master thesis was to set up a machine learning Neural Network based on a transport problem. In the project, a comprehensive state of the art is done in order to study which methods exist and are used nowadays, both in the transport area and in the machine learning area. It has been seen that in transport modelling, activity-based models are generally better suited for solving the numerous kinds of problems there might be, because they incorporate more detailed and variable information at a disaggregate level like the trip purpose, location, time, transport mode, etc. For this reason, they are more able to explain social behaviors and how people may react to new policies or external inputs like weather, change of plans, terrorist attacks or natural catastrophes, etc.

On the other hand, the state of the art on Graph Neural Networks (GNNs) shows that there has been a huge development in the last 10 years. The recent success of GNNs has boosted research on data mining and pattern recognition, being able to develop tasks such as object detection, machine translation, image recognition and generation, and speech recognition that once relied in handcrafted feature engineering models to extract informative feature sets.

On this basis, two deep learning models are developed and analyzed in order to bring the two worlds together and try to solve some classical transport problems with the new approach of Neural Networks.

The first model one consists in a Variational Autoencoder (VAE), which recently has become one of the most popular ways to unsupervised learning of complicated distributions. VAEs overcome the main drawbacks that generative models have suffered in the last years since backpropagation allows them to be trained in a fast way and also the assumptions they made in the model are relatively weak, which gives them a great flexibility to fit numerous kinds of data. In this line, a VAE has been defined in order to deal with activity-based sequences of data, which represented 13 different activities that individuals carried out during a day. The main objective was to capture the nature of the data and recognize patterns among it. After that, the VAE has been set to generate new sequences of data based on the encoded features on the latent space. To show VAEs potential, a simpler model based on a Frequency Analysis (FA) of the population from the dataset has been created too. The results presented in sections 6.2.2.4 and 6.2.3 show that the VAE is a very powerful and useful tool to deal with these activity sequences, since the new generated sequences are very close to the original ones and also, they are ordered and positioned in time in an appropriate way, clearly outperforming the FA model.

The second model consists in a Long Short-Term Memory (LSTM) encoder-decoder architecture designed for sequence to sequence prediction. One of the examples that inspired this model are the neural networks dedicated to machine translation from sentences of one language to sentences from another language. The model is to be trained in a way to find some hidden correlations about an input sequence and an output sequence, in a way that when a new unseen sequence of data is presented, it is able to predict the sequence that will come next. In this line, we created a model that is set to predict the next Y timesteps according to the information provided by an input sequence of the previous X timesteps. However, early results show that predictions are not being very accurate. Also, there had not been more time to continue developing this model for various reasons, but this first results show that this is a line of work where there can be a very interesting further development, since it will allow traffic planners to have more and better information about social mobility.

All in all, the personal objectives of this project have been achieved. First, I have been able to discover the state of the art in transport modeling, as well as the main models that are being used nowadays, together with the importance of activity-based models. Second, I have been able to introduce myself into the fascinating world of machine learning and to visualize all the potential this new data mining field offers.

7.2 Further Research

Although the proposed goals and the exposed objectives have been achieved, this project opens some further research lines to extend the obtained results. This section exposes some of the most interesting ones.

First of all, the use of a Sequence Alignment Method (SAM) could be used in order to compare the activity sequences, the original one and the new one, whether it has been completely generated by the Variational Autoencoder or it has been predicted from certain point by the LSTM model. SAMs are very used in bioinformatics in order to arrange sequences of DNA, RNA or protein. In non-biological sequences, SAMs are used to calculate the distance cost between characters, for example. They are also used to align long, highly variable sequences that cannot be aligned manually. It could be especially useful in the LSTM part, to help or "force" the predicted sequences be more similar to the original one.

Secondly, the sequences generated by the Variational Autoencoder could be used for agent-based modeling (ABM). These models are used to simulate actions and interactions between autonomous agents (could be persons, companies, organizations, groups, etc.) with the aim to observe their effects on a system. In transport, they are used, for example, to explore daily commuting. Balmer et al. (2008) developed an agent-based model based on the toolkit MATSim-T to model traffic demand and traffic flow and their interactions in the Greater Zürich Area.

And third, the even though the LSTM model developed in this project has not achieved very good results, with more investigation and time, it could deliver very interesting results that may be worth the effort. The fact of being able to predict activity sequences could be very useful for transport planers to know people's mobility and transport demand. In addition, it could serve not only for transport related purposes, but also to know demand on other activities, like shopping or taking a meal outside.

References

Abrantes, P. A. L., & Wardman, M. R. (2011). Meta-analysis of UK values of travel time: An update. *Transportation Research Part A: Policy and Practice*, *45*(1), 1–17. https://doi.org/10.1016/j.tra.2010.08.003

Anda, C., Erath, A., & Fourie, P. J. (2017). Transport modelling in the age of big data. *International Journal of Urban Sciences*, *21*(sup1), 19–42. https://doi.org/10.1080/12265934.2017.1281150

Arentze, T. A., Harry, & Timmermans, J. P. (2000). *1 Albatross – a Learning-Based Transportation Oriented Simulation System*.

Asif, M. T., Dauwels, J., Goh, C. Y., Oran, A., Fathi, E., Xu, M., Mohan, D., Mitrovic, N., & Jaillet, P. (2013). *Spatiotemporal Patterns in Large-Scale Traffic Speed Prediction*. ResearchGate. https://www.researchgate.net/publication/259624533_Spatiotemporal_Patterns_in_Large-Scale_Traffic_Speed_Prediction

Athira, I. C., Muneera, C. P., Krishnamurthy, K., & Anjaneyulu, M. V. L. R. (2016). Estimation of Value of Travel Time for Work Trips. *Transportation Research Procedia*, *17*, 116–123. https://doi.org/10.1016/j.trpro.2016.11.067

Atwood, J., & Towsley, D. F. (2016). Diffusion-Convolutional Neural Networks. *NIPS*.

Balmer, M., Meister, K., Rieser, M., Nagel, K., & Axhausen, K. W. (2008). *Agent-based simulation of travel demand: Structure and computational performance of MATSim-T* [Application/pdf]. 37 p. https://doi.org/10.3929/ETHZ-A-005626451

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. *ArXiv:1206.5533 [Cs]*. http://arxiv.org/abs/1206.5533

Blaauw, F., & Emerencia, o. (2016). *Deep learning the beautiful mind | Mindwise*. https://mindwise-groningen.nl/deep-learning-the-beautiful-mind/

Brownlee, J. (2017a, July 27). Why One-Hot Encode Data in Machine Learning? *Machine Learning Mastery*. https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/

Brownlee, J. (2017b, October 25). How to Develop a Seq2Seq Model for Neural Machine Translation in Keras. *Machine Learning Mastery*. https://machinelearningmastery.com/define-encoder-decoder-sequence-sequence-model-neural-machine-translation-keras/

Brownlee, J. (2017c, November 1). How to Develop an Encoder-Decoder Model for Sequence-to-Sequence Prediction in Keras. *Machine Learning Mastery*. https://machinelearningmastery.com/develop-encoder-decoder-model-sequence-sequence-prediction-keras/

Brownlee, J. (2018, July 19). Difference Between a Batch and an Epoch in a Neural Network. *Machine Learning Mastery*. https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/

Brownlee, J. (2019a, January 20). How to Control the Stability of Training Neural Networks With the Batch Size. *Machine Learning Mastery*. https://machinelearningmastery.com/how-to-control-the-speed-and-stability-of-training-neural-networks-with-gradient-descent-batch-size/

Brownlee, J. (2019b, April 18). A Gentle Introduction to Padding and Stride for Convolutional Neural Networks. *Machine Learning Mastery*. https://machinelearningmastery.com/padding-and-stride-for-convolutional-neural-networks/

Bruna, J., Zaremba, W., Szlam, A., & LeCun, Y. (2014). Spectral Networks and Locally Connected Networks on Graphs. *ArXiv:1312.6203 [Cs]*. http://arxiv.org/abs/1312.6203

Cai, P., Wang, Y., Lu, G., Chen, P., Ding, C., & Sun, J. (2016). *A spatiotemporal correlative k-nearest neighbor model for short-term traffic multistep forecasting*. ResearchGate. https://www.researchgate.net/publication/285459003_A_spatiotemporal_correlative_k-nearest_neighbor_model_for_short-term_traffic_multistep_forecasting

Cao, S., Lu, W., & Xu, Q. (2016). Deep neural networks for learning graph representations. *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 1145–1152.

Castiglione, Joe, B., Mark, & Gliebe, John. (2015). *Activity-Based Travel Demand Models: A Primer*. https://doi.org/10.17226/22357

Chen, Jianfei, Zhu, J., & Song, L. (2018). Stochastic Training of Graph Convolutional Networks with Variance Reduction. *ArXiv:1710.10568 [Cs, Stat]*. http://arxiv.org/abs/1710.10568

Chen, Jie, Ma, T., & Xiao, C. (2018). FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. *ArXiv:1801.10247 [Cs]*. http://arxiv.org/abs/1801.10247

Chen, S., Varma, R., Sandryhaila, A., & Kovačević, J. (2015). Discrete Signal Processing on Graphs: Sampling Theory. *IEEE Transactions on Signal Processing*, *63*(24), 6510–6523. https://doi.org/10.1109/TSP.2015.2469645

Chiang, W.-L., Liu, X., Si, S., Li, Y., Bengio, S., & Hsieh, C.-J. (2019). Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 257–266. https://doi.org/10.1145/3292500.3330925

Chien, S. I. J., Liu, X., & Ozbay, K. (2003). Predicting Travel Times for the South Jersey Real-Time Motorist Information System. *Transportation Research Record*, *1855*(1), 32–40. https://doi.org/10.3141/1855-04

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *ArXiv:1406.1078 [Cs, Stat]*. http://arxiv.org/abs/1406.1078

Chollet, F. (2017). *A ten-minute introduction to sequence-to-sequence learning in Keras*. https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html

Cools, M., Brijs, K., Tormans, H., Moons, E., Janssens, D., & Wets, G. (2011). The socio-cognitive links between road pricing acceptability and changes in travel-behavior. *Transportation Research Part A: Policy & Practice*, *45*(8), 779–788. https://doi.org/10.1016/j.tra.2011.06.006

Cools, M., & Creemers, L. (2013). The dual role of weather forecasts on changes in activity-travel behavior. *Journal of Transport Geography*, *28*, 167–175. https://doi.org/10.1016/j.jtrangeo.2012.11.002

Cornelis, E., Hubert, M., Huynen, P., Lebrun, K., Patriarche, G., De Witte, A., Creemers, L., Declercq, K., Janssens, D., Castaigne, M., Hollaert, L., & Walle, F. (2010). *La mobilité en Belgique en 2010: Résultats de l'enquête BELDAM*.

Creemers, L., Wets, G., & Cools, M. (2015). Meteorological variation in daily travel behaviour: Evidence from revealed preference data from the Netherlands. *Theoretical & Applied Climatology*, *120*(1–2), 183–194. https://doi.org/10.1007/s00704-014-1169-0

Cui, Z., Ke, R., Pu, Z., & Wang, Y. (2020). Stacked Bidirectional and Unidirectional LSTM Recurrent Neural Network for Forecasting Network-wide Traffic State with Missing Values. *ArXiv:2005.11627 [Cs, Eess, Stat]*. http://arxiv.org/abs/2005.11627

Dai, H., Kozareva, Z., Dai, B., Smola, A. J., & Song, L. (2018). *Learning Steady-States of Iterative Algorithms over Graphs*. 9.

Defferrard, M., Bresson, X., & Vandergheynst, P. (2017). Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. *ArXiv:1606.09375 [Cs, Stat]*. http://arxiv.org/abs/1606.09375

DelSole, M. (2018, April 24). *What is One Hot Encoding and How to Do It*. Medium. https://medium.com/@michaeldelsole/what-is-one-hot-encoding-and-how-to-do-it-f0ae272f1179

Dijst, M., & Vidakovic, V. (1997). *Activity-Based Approaches to Travel Analysis*. 117–134.

Doersch, C. (2016). Tutorial on Variational Autoencoders. *ArXiv:1606.05908 [Cs, Stat]*. http://arxiv.org/abs/1606.05908

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, *14*(2), 179–211. https://doi.org/10.1016/0364-0213(90)90002-E

Ettema, D., Borgers, A., & Timmermans, H. (1995). *Competing risk hazard model of activity choice, timing, sequencing, and duration.* https://scholar.google.com/scholar_lookup?hl=en&publication_year=1995&pages=101-109&author=D.F+Ettemaauthor=A.+W.+J.+Borgers&author=H.+J.+P.+Timmermans&title=Competing+risk+hazard+model+of+activity+choice%2C+timing%2C+sequencing+and+duration

Ettema, D., Borgers, A., & Timmermans, H. (1996). SMASH (Simulation Model of Activity Scheduling Heuristics): Some Simulations. *Transportation Research Record: Journal of the Transportation Research Board*, *1551*(1), 88–94. https://doi.org/10.1177/0361198196155100112

Ferrer, S., & Ruiz, T. (2014). Factors influencing the travel scheduling of driving trips of habitual car users. *Transportation Research Record*, *2412*, 100–108. Scopus. https://doi.org/10.3141/2412-12

Fricker, Jon D., & Whitford, Robert K. (2003). *Fundamentals of Transportation Engineering: A Multimodal Systems Approach: Amazon.es: Fricker, Jon D., Whitford, Robert K.: Libros en idiomas extranjeros*. https://www.amazon.es/Fundamentals-Transportation-Engineering-Multimodal-Approach/dp/0130351245

Friedrich, M. (2007). *Multimodal Transport Planning and Modelling | Institute for Road and Transport Science | University of Stuttgart*. https://www.isv.uni-stuttgart.de/en/vuv/courses/multimodal/

Gallicchio, C., & Micheli, A. (2010). Graph Echo State Networks. *The 2010 International Joint Conference on Neural Networks (IJCNN)*. https://doi.org/10.1109/IJCNN.2010.5596796

Gärling, T., Kwan, M.-P., & Golledge, R. G. (1994). Computational-process modelling of household activity scheduling. *Transportation Research Part B: Methodological*, *28*(5), 355–364. https://doi.org/10.1016/0191-2615(94)90034-5

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural Message Passing for Quantum Chemistry. *ArXiv:1704.01212 [Cs]*. http://arxiv.org/abs/1704.01212

Gori, M., Monfardini, G., & Scarselli, F. (2005). A new model for learning in graph domains. *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* https://doi.org/10.1109/IJCNN.2005.1555942

Graves, A. (2008). *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer-Verlag. https://doi.org/10.1007/978-3-642-24797-2

Hamilton, W. L., Ying, R., & Leskovec, J. (2018). Inductive Representation Learning on Large Graphs. *ArXiv:1706.02216 [Cs, Stat]*. http://arxiv.org/abs/1706.02216

Henaff, M., Bruna, J., & LeCun, Y. (2015). Deep Convolutional Networks on Graph-Structured Data. *ArXiv:1506.05163 [Cs]*. http://arxiv.org/abs/1506.05163

Henson, K., Goulias, K., & Golledge, R. (2009). An assessment of activity-based modeling and simulation for applications in operational studies, disaster preparedness, and homeland security. *Transportation Letters*, *1*(1), 19–39. https://doi.org/10.3328/TL.2009.01.01.19-39

Hinton, G., Deng, L., Yu, D., Dahl, G., Mohamed, A., Jaitly, N., Vanhoucke, V., Nguyen, P., Sainath, T., & Kingsbury, B. (2012). *Deep Neural Networks for Acoustic Modeling in Speech Recognition*. 27.

Hinton, G., & Sejnowski, T. J. (1999). *Unsupervised Learning | The MIT Press*. The MIT Press. https://mitpress.mit.edu/books/unsupervised-learning

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Jain, A., Zamir, A. R., Savarese, S., & Saxena, A. (2016). Structural-RNN: Deep Learning on Spatio-Temporal Graphs. *ArXiv:1511.05298 [Cs]*. http://arxiv.org/abs/1511.05298

Jeong, J. (2019, July 17). *The Most Intuitive and Easiest Guide for CNN*. Medium. https://towardsdatascience.com/the-most-intuitive-and-easiest-guide-for-convolutional-neural-network-3607be47480

Joglekar, S. (2017, February 3). How Neural Networks generate Visual Art from inspiration. *Sachin Joglekar's Blog*. https://codesachin.wordpress.com/2017/02/03/how-neural-networks-generate-visual-art-from-inspiration/

Jordan, M. I. (1986). *SERIAL ORDER: A PARALLEL DISTRmUTED PROCESSING APPROACH*. 46.

Karlaftis, M. G., & Vlahogianni, E. I. (2011). Statistical methods versus neural networks in transportation research: Differences, similarities and some insights. *Transportation Research Part C: Emerging Technologies*, *19*(3), 387–399.

Kipf, T. N., & Welling, M. (2016). Variational Graph Auto-Encoders. *ArXiv:1611.07308 [Cs, Stat]*. http://arxiv.org/abs/1611.07308

Kipf, T. N., & Welling, M. (2017). Semi-Supervised Classification with Graph Convolutional Networks. *ArXiv:1609.02907 [Cs, Stat]*. http://arxiv.org/abs/1609.02907

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90. https://doi.org/10.1145/3065386

Kwan, M.-P. (1997). GISICAS: AN ACTIVITY-BASED TRAVEL DECISION SUPPORT SYSTEM USING A GIS-INTERFACED COMPUTATIONAL-PROCESS MODEL. *ACTIVITY-BASED APPROACHES TO TRAVEL ANALYSIS*. https://trid.trb.org/view/570134

Lecun, Y., & Bengio, Y. (1995). Convolutional networks for images, speech, and time-series. *The Handbook of Brain Theory and Neural Networks*.

https://nyuscholars.nyu.edu/en/publications/convolutional-networks-for-images-speech-and-time-series

Levie, R., Monti, F., Bresson, X., & Bronstein, M. M. (2018). CayleyNets: Graph Convolutional Neural Networks with Complex Rational Spectral Filters. *ArXiv:1705.07664 [Cs]*. http://arxiv.org/abs/1705.07664

Li, Yaguang, Yu, R., Shahabi, C., & Liu, Y. (2018). Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting. *ArXiv:1707.01926 [Cs, Stat]*. http://arxiv.org/abs/1707.01926

Li, Yujia, Tarlow, D., Brockschmidt, M., & Zemel, R. (2017). Gated Graph Sequence Neural Networks. *ArXiv:1511.05493 [Cs, Stat]*. http://arxiv.org/abs/1511.05493

Li, Yujia, Vinyals, O., Dyer, C., Pascanu, R., & Battaglia, P. (2018). Learning Deep Generative Models of Graphs. *ArXiv:1803.03324 [Cs, Stat]*. http://arxiv.org/abs/1803.03324

Litman, T. (2002). *Transportation Cost and Benefit Analysis: Techniques, Estimates and Implications*.

Luong, M.-T., Pham, H., & Manning, C. D. (2015). Effective Approaches to Attention-based Neural Machine Translation. *ArXiv:1508.04025 [Cs]*. http://arxiv.org/abs/1508.04025

Ma, X., Dai, Z., He, Z., Na, J., Wang, Y., & Wang, Y. (2017). Learning Traffic as Images: A Deep Convolutional Neural Network for Large-Scale Transportation Network Speed Prediction. *ArXiv:1701.04245 [Cs, Stat]*. http://arxiv.org/abs/1701.04245

Ma, X., Tao, Z., Wang, Y., Yu, H., & Wang, Y. (2015). Long short-term memory neural network for traffic speed prediction using remote microwave sensor data. *Transportation Research Part C: Emerging Technologies*, *54*(0). https://trid.trb.org/view/1350260

Mars, L., & Ruiz, T. (2018). Determinants of elimination decisions in the activity scheduling process. *Transportation Letters*, *10*(4), 185–201. Scopus. https://doi.org/10.1080/19427867.2016.1242882

Masters, D., & Luschi, C. (2018). Revisiting Small Batch Training for Deep Neural Networks. *ArXiv:1804.07612 [Cs, Stat]*. http://arxiv.org/abs/1804.07612

McNALLY, M. (2000). *The Four Step Model*.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., & Teller, E. (1953). Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, *21*(6), 1087–1092. https://doi.org/10.1063/1.1699114

Micheli, A., Sona, D., & Sperduti, A. (2004). Contextual processing of structured data by recursive cascade correlation. *IEEE Transactions on Neural Networks*, *15*(6), 1396–1410. https://doi.org/10.1109/TNN.2004.837783

Micheli, Alessio. (2009). Neural Network for Graphs: A Contextual Constructive Approach. *IEEE Transactions on Neural Networks*, *20*(3), 498–511. https://doi.org/10.1109/TNN.2008.2010350

Mohr, F. (2017, November 9). *Teaching a Variational Autoencoder (VAE) to draw MNIST characters*. Medium. https://towardsdatascience.com/teaching-a-variational-autoencoder-vae-to-draw-mnist-characters-978675c95776

Niepert, M., Ahmed, M., & Kutzkov, K. (2016). Learning Convolutional Neural Networks for Graphs. *ArXiv:1605.05273 [Cs, Stat]*. http://arxiv.org/abs/1605.05273

Olah, C. (2015). *Understanding LSTM Networks*. http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Pascanu, R., Mikolov, T., & Bengio, Y. (2013). On the difficulty of training recurrent neural networks. *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, III–1310–III–1318.

Pearlmutter, B. A. (1989). *(PDF) Learning State Space Trajectories in Recurrent Neural Networks*. ResearchGate. https://www.researchgate.net/publication/238833309_Learning_State_Space_Trajectories_in_Recurrent_Neural_Networks

Pineda, F. J. (1987). Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, *59*(19), 2229–2232. https://doi.org/10.1103/PhysRevLett.59.2229

Puig-Arnavat, M., & Bruno, J. C. (2015). *Backpropagation Algorithm—An overview | ScienceDirect Topics*. https://www.sciencedirect.com/topics/engineering/backpropagation-algorithm

Pykes, K. (2020, May 17). *The Vanishing/Exploding Gradient Problem in Deep Neural Networks*. Medium. https://towardsdatascience.com/the-vanishing-exploding-gradient-problem-in-deep-neural-networks-191358470c11

Rasouli, S., & Timmermans, H. (2014). Activity-based models of travel demand: Promises, progress and prospects. *International Journal of Urban Sciences*, *18*(1), 31–60. https://doi.org/10.1080/12265934.2013.835118

Recker, W. W. (1995). The household activity pattern problem: General formulation and solution. *Transportation Research Part B: Methodological*, *29*(1), 61–77. https://doi.org/10.1016/0191-2615(94)00023-S

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. *ArXiv:1506.02640 [Cs]*. http://arxiv.org/abs/1506.02640

Ruiz, T., & Timmermans, H. (2006). Changing the timing of activities in resolving scheduling conflicts. *Transportation*, *33*(5), 429–445. Scopus. https://doi.org/10.1007/s11116-006-0010-8

Ruiz, T., & Timmermans, H. (2008). Changing the duration of activities in resolving scheduling conflicts. *Transportation Research Part A: Policy and Practice*, *42*(2), 347–359. Scopus. https://doi.org/10.1016/j.tra.2007.10.007

Saadi, I., Mustafa, A., Teller, J., & Cools, M. (2016). Forecasting travel behavior using Markov Chains-based approaches. *Transportation Research: Part C*, *69*, 402–417. https://doi.org/10.1016/j.trc.2016.06.020

Saadi, I., Mustafa, A., Teller, J., & Cools, M. (2018). Investigating the impact of river floods on travel demand based on an agent-based modeling approach: The case of Liège, Belgium. *Transport Policy*, *67*, 102–110. https://doi.org/10.1016/j.tranpol.2017.09.009

Saadi, I., Mustafa, A., Teller, J., Farooq, B., & Cools, M. (2016). Hidden Markov Model-based population synthesis. *Transportation Research: Part B*, *90*, 1–21. https://doi.org/10.1016/j.trb.2016.04.007

Sandryhaila, A., & Moura, J. M. F. (2013). Discrete Signal Processing on Graphs. *IEEE Transactions on Signal Processing*, *61*(7), 1644–1656. https://doi.org/10.1109/TSP.2013.2238935

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*.

Seo, Y., Defferrard, M., Vandergheynst, P., & Bresson, X. (2016). Structured Sequence Modeling with Graph Convolutional Recurrent Networks. *ArXiv:1612.07659 [Cs, Stat]*. http://arxiv.org/abs/1612.07659

Sharma, S. (2019, February 14). *Activation Functions in Neural Networks*. Medium. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

Sherstinky, A. (2018). *(PDF) Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network*. ResearchGate. https://www.researchgate.net/publication/326988050_Fundamentals_of_Recurrent_Neural_Network_RNN_and_Long_Short-Term_Memory_LSTM_Network

Shuman, D. I., Narang, S. K., Frossard, P., Ortega, A., & Vandergheynst, P. (2013). The Emerging Field of Signal Processing on Graphs: Extending High-Dimensional Data Analysis to Networks and Other Irregular Domains. *IEEE Signal Processing Magazine*, *30*(3), 83–98. https://doi.org/10.1109/MSP.2012.2235192

Simonovsky, M., & Komodakis, N. (2018). GraphVAE: Towards Generation of Small Graphs Using Variational Autoencoders. *ArXiv:1802.03480 [Cs]*. http://arxiv.org/abs/1802.03480

Sperduti, A., & Starita, A. (1997). Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, *8*(3), 714–735. https://doi.org/10.1109/72.572108

Tan, H., Wu, Y., Shen, B., Jin, P. J., & Ran, B. (2016). Short-Term Traffic Prediction Based on Dynamic Tensor Completion. *IEEE Transactions on Intelligent Transportation Systems*, *17*(8). https://trid.trb.org/view/1422586

Torres, J. (2012). *Master in Innovation and Research in Informatics | FIB - Barcelona School of Informatics*. https://www.fib.upc.edu/en/studies/masters/master-innovation-and-research-informatics

Tu, K., Cui, P., Wang, X., Yu, P. S., & Zhu, W. (2018). Deep Recursive Network Embedding with Regular Equivalence. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2357–2366. https://doi.org/10.1145/3219819.3220068

Vincent, P., Larochelle, H., Bengio, Y., & Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. *Proceedings of the 25th International Conference on Machine Learning*, 1096–1103. https://doi.org/10.1145/1390156.1390294

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., & Manzagol, P.-A. (2010). *Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion*. 38.

Wang, D., Cui, P., & Zhu, W. (2016). Structural Deep Network Embedding. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1225–1234. https://doi.org/10.1145/2939672.2939753

Wei, W., Wu, H., & Ma, H. (2019). An AutoEncoder and LSTM-Based Traffic Flow Prediction Method. *Sensors*, *19*(13), 2946. https://doi.org/10.3390/s19132946

Wu, Yonghui, Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, Ł., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., … Dean, J. (2016). Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *ArXiv:1609.08144 [Cs]*. http://arxiv.org/abs/1609.08144

Wu, Yuankai, & Tan, H. (2016). Short-term traffic flow forecasting with spatial-temporal correlation in a hybrid deep learning framework. *ArXiv:1612.01022 [Cs]*. http://arxiv.org/abs/1612.01022

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Yu, P. S. (2019). A Comprehensive Survey on Graph Neural Networks. *ArXiv:1901.00596 [Cs, Stat]*. http://arxiv.org/abs/1901.00596

Yu, B., Yin, H., & Zhu, Z. (2018). *Spatio-Temporal Graph Convolutional Networks: A Deep Learning Framework for Traffic Forecasting*. 3634–3640.

Yu, H., Wu, Z., Wang, S., Wang, Y., & Ma, X. (2017). Spatiotemporal Recurrent Convolutional Networks for Traffic Prediction in Transportation Networks. *ArXiv:1705.02699 [Cs]*. http://arxiv.org/abs/1705.02699

Zhang, J., Shi, X., Xie, J., Ma, H., King, I., & Yeung, D.-Y. (2018). GaAN: Gated Attention Networks for Learning on Large and Spatiotemporal Graphs. *ArXiv:1803.07294 [Cs]*. http://arxiv.org/abs/1803.07294

Zhao, Z., Chen, W., Wu, X., Chen, P. C. Y., & Liu, J. (2017). *LSTM network: A deep learning approach for short-term traffic forecast*. https://doi.org/10.1049/IET-ITS.2016.0208

Appendices

Code for the Frequency Analysis model

```
import pandas as pd
from datetime import timedelta
import numpy as np

# load the data
df_yg = pd.read_csv("dataset_belgium/merged.csv", sep=';')

# df_wkd = df.loc[(df.weekday > 1.0) & (df.weekday < 7.0)]
# df_yg = df_wkd.loc[(df.age < 65.0)]
del df_yg['g_regi']
del df_yg['age']
del df_yg['mainmode']
del df_yg['t_withchild']
del df_yg['weekday']
del df_yg['h_hierarchie_urbaine']

df_yg = df_yg.transpose()

df_yg = df_yg[1:]
timeline = []
for timestep in range(0,len(df_yg.index)):
    time = timedelta(seconds=timestep*2*60)
    time = str(time)
    timeline.append(time)

td = pd.TimedeltaIndex(timeline, unit = 'm', freq = 'infer', name =
'timeline')
df_yg = df_yg.set_index(td)

df_t = df_yg.transpose()
train_size = int(df_t.shape[0]*0.8)

df = df_t[:train_size]

d = {}
numbers = [0,1,2,3,4,5,6,7,8,9,10,11,12,13]
for column in df:
    probs = df[column].value_counts(normalize = True)
    l = []
    for number in numbers:
        if number in probs:
            t = (number, probs[number])
            l.append(t)
        else:
            t = (number, 0.0)
            l.append(t)
    d[column] = l

def generate_samples(l, n_samples):
    weights = []
    for elem in l:
        weights.append(elem[1])
```

```
    samples = np.random.choice(14, n_samples, p = weights)
    return samples

data = {}
n_samples = 1375
for key in d.keys():
    samples = generate_samples(d[key], n_samples)
    data[key] = samples

freq_df = pd.DataFrame(data)
freq_df.to_csv("dataset_belgium/frequency_analysis_all_population.csv"
)

def trips_per_day(serie):
    l = serie.tolist()
    count = 0
    for i, value in enumerate(l):
        if l[i] == 13 and len(l) == 0:
            count += 1
        elif l[i] == 13 and l[i-1] != 13:
            count += 1
    return count

def percentage_travel_per_day(serie):
    se = serie.value_counts(normalize=True)
    if 13 in se.index:
        return se.loc[13]*100
    else:
        return 0

def n_activities_outside(serie):
#only takes into account the activity one time
    se = list(serie.unique())
    if 13 in se:
        se.remove(13)
    if 2 in se:
        se.remove(2)
    return len(se)

def n_hours_out_home(serie):
    l = serie.tolist()
    ll = [i for i in l if i != 2]
    hours = (len(ll)*2)/60
    return hours

def percentage_hours_out_home(serie):
    l = serie.tolist()
    ll = [i for i in l if i != 2]
    hours = (len(ll)*2)/60
    p = (hours/24)*100
    return p

def n_hours_at_work(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 3]
    hours = len(ll)*2/60
    return hours

def percentage_hours_at_work(serie):
```

```
    l = serie.tolist()
    ll = [i for i in l if i == 3]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_at_home(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 2]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_work_trip(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 4]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_courses(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 5]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_eat_outside(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 6]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_shopping(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 7]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_services(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 8]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_family_visit(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 9]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_walk_tour(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 10]
    hours = len(ll)*2/60
```

```
    p = (hours/24)*100
    return p

def percentage_hours_sport_leisure(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 11]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

analysed_data = []

for index in freq_df.index:
    di = {}

    di['n_trips_day_in'] = trips_per_day(freq_df.iloc[index])

    di['percentage_travel_day_in'] =
percentage_travel_per_day(freq_df.iloc[index])

    di['n_activities_outside_in'] =
n_activities_outside(freq_df.iloc[index])

    di['percentage_hours_outside_in'] =
percentage_hours_out_home(freq_df.iloc[index])

    di['percentage_hours_at_work_in'] =
percentage_hours_at_work(freq_df.iloc[index])

    di['percentage_hours_at_home_in'] =
percentage_hours_at_home(freq_df.iloc[index])

    di['percentage_hours_work_trip_in'] =
percentage_hours_work_trip(freq_df.iloc[index])

    di['percentage_hours_courses_in'] =
percentage_hours_courses(freq_df.iloc[index])

    di['percentage_hours_eat_outside_in'] =
percentage_hours_eat_outside(freq_df.iloc[index])

    di['percentage_hours_shopping_in'] =
percentage_hours_shopping(freq_df.iloc[index])

    di['percentage_hours_services_in'] =
percentage_hours_services(freq_df.iloc[index])

    di['percentage_hours_family_visit_in'] =
percentage_hours_family_visit(freq_df.iloc[index])

    di['percentage_hours_walk_tour_in'] =
percentage_hours_walk_tour(freq_df.iloc[index])

    di['percentage_hours_sport_leisure_in'] =
percentage_hours_sport_leisure(freq_df.iloc[index])

    analysed_data.append(di)
```

```
freq_analysis_df = pd.DataFrame(analysed_data)

freq_analysis_df.to_csv("dataset_belgium/freq_analysis_metrics_all.csv
")
```

Code for the Variational Autoencoder model

```
#with timesteps of 2 mins
#not binary, one-hot-encoded
#all data (not deleting weekends and retired people)


import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import pandas as pd
from datetime import timedelta
from numpy import array
from sklearn.preprocessing import OneHotEncoder

"""## Create a sampling layer"""

class Sampling(layers.Layer):
  """Uses (z_mean, z_log_var) to sample z, the vector encoding a
sequence."""

  def call(self, inputs):
    z_mean, z_log_var = inputs
    batch = tf.shape(z_mean)[0]
    dim = tf.shape(z_mean)[1]
    epsilon = tf.keras.backend.random_normal(shape=(batch, dim))
    return z_mean + tf.exp(0.5 * z_log_var) * epsilon

"""## Build the encoder"""

#tuning parameters
latent_dim = 12
EPOCHS = 30
BATCH_SIZE = 64
TRAIN_SPLIT = 0.8

encoder_inputs = keras.Input(shape=(720, 14))
x = layers.Conv1D(64, 3, activation='relu', strides=1,
padding='same')(encoder_inputs)
x = layers.Conv1D(32, 3, activation='relu', strides=1,
padding='same')(x)
x = layers.Flatten()(x)
x = layers.Dense(32, activation='relu')(x)
z_mean = layers.Dense(latent_dim, name='z_mean')(x)
z_log_var = layers.Dense(latent_dim, name='z_log_var')(x)
z = Sampling()([z_mean, z_log_var])
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var, z],
name='encoder')
encoder.summary()

"""## Build the decoder"""


latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(720 * 32, activation='relu')(latent_inputs)
x = layers.Reshape((720, 32))(x)
```

```
x = layers.Conv1D(32, 3, activation='relu', strides=1,
padding='same')(x)
x = layers.Conv1D(64, 3, activation='relu', strides=1,
padding='same')(x)
decoder_outputs = layers.Conv1D(14, 3, activation='softmax',
padding='same')(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name='decoder')
decoder.summary()

"""## Define the VAE as a `Model` with a custom `train_step`"""

class VAE(keras.Model):

    def __init__(self, encoder, decoder, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.probas = keras.layers.Dense(10, activation="softmax")

    @tf.function(input_signature=[
        tf.TensorSpec(shape=[None], dtype=tf.string)])

    def serve(self, serialized):
        expected_features = {
            "image": tf.io.FixedLenFeature([28 * 28],
dtype=tf.float32)
        }
        examples = tf.io.parse_example(serialized, expected_features)
        return self.probas(examples["image"])

    def call(self, inputs):
        return self.probas(inputs)

    def train_step(self, data):
        data = data[0]
        with tf.GradientTape() as tape:
          z_mean, z_log_var, z = encoder(data)
          reconstruction = decoder(z)
          reconstruction_loss =
tf.reduce_mean(keras.losses.categorical_crossentropy(data,
reconstruction))
          reconstruction_loss *= 720
          kl_loss = 1 + z_log_var - tf.square(z_mean) -
tf.exp(z_log_var)
          kl_loss = tf.reduce_mean(kl_loss)
          kl_loss *= -0.5
          total_loss = reconstruction_loss + kl_loss
        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads,
self.trainable_weights))
        return {'loss': total_loss,
            'reconstruction_loss': reconstruction_loss,
            'kl_loss': kl_loss}

"""## Train the VAE"""
# load the data
df_yg = pd.read_csv("dataset_belgium/merged.csv", sep=';')

# df_wkd = df.loc[(df.weekday > 1.0) & (df.weekday < 7.0)]
```

```
# df_yg = df_wkd.loc[(df.age < 65.0)]
del df_yg['g_regi']
del df_yg['age']
del df_yg['mainmode']
del df_yg['t_withchild']
del df_yg['weekday']
del df_yg['h_hierarchie_urbaine']

df_yg = df_yg.transpose()

df_yg = df_yg[1:]
timeline = []
for timestep in range(0,len(df_yg.index)):
    time = timedelta(seconds=timestep*2*60)
    time = str(time)
    timeline.append(time)

td = pd.TimedeltaIndex(timeline, unit = 'm', freq = 'infer', name =
'timeline')
df_yg = df_yg.set_index(td)

df = df_yg.transpose()

X = []
for index in df.index:
    X.append(df.loc[index])

dta = array(X)
X = np.expand_dims(dta, axis=2)
train_len = int(TRAIN_SPLIT * len(X))
x_train_b, x_test_b = X[:train_len], X[train_len:]

x_train_flat = x_train_b.reshape(-1)
x_test_flat = x_test_b.reshape(-1)


nb_classes = 14

def indices_to_one_hot(data, nb_classes):
    return np.eye(nb_classes)[data]

xt1 = indices_to_one_hot(x_train_flat, nb_classes)
x_train =
xt1.reshape(x_train_b.shape[0],x_train_b.shape[1],nb_classes)

xt2 = indices_to_one_hot (x_test_flat, nb_classes)
x_test = xt2.reshape(x_test_b.shape[0],x_test_b.shape[1],nb_classes)

# train
vae = VAE(encoder, decoder)
vae.compile(optimizer=keras.optimizers.Adam(), run_eagerly=True)
vae.fit(x_train, epochs=EPOCHS, batch_size=BATCH_SIZE)

#infer data
encoded_seqs = encoder.predict(x_test)
decoded_seqs = decoder.predict(encoded_seqs)

#pass decoded_seqs to a dataframe comparable to the original
```

```
td = pd.TimedeltaIndex(timeline, unit = 'm', freq = 'infer', name =
'timeline')
td = td.to_frame()
x_test_df = td.transpose()
decoded_df = td.transpose()
x_test_df = x_test_df[1:]
decoded_df = decoded_df[1:]

#undo one-hot encoding
x_test_undone = np.argmax(x_test, axis=2)
decoded_seqs_undone = np.argmax(decoded_seqs, axis=2)
x_test_final = x_test_undone.reshape(x_test.shape[0],
x_test.shape[1],1)
decoded_seqs_final =
decoded_seqs_undone.reshape(decoded_seqs.shape[0],
decoded_seqs.shape[1],1)

#insert in the new dataframes
for i, seq in enumerate(x_test_final):
    seq = seq.flatten()
    x_test_df.loc[i] = seq

for i, seq in enumerate(decoded_seqs_final):
    seq = seq.flatten()
    decoded_df.loc[i] = seq

x_test_df.to_csv("dataset_belgium/VAE_m42v3/x_test.csv")
decoded_df.to_csv("dataset_belgium/VAE_m42v3/decoded.csv")

x_test_df = pd.read_csv("dataset_belgium/VAE_m42v3/x_test.csv")
x_test_df = x_test_df.transpose()
x_test_df = x_test_df[1:]
x_test_df = x_test_df.transpose()
# x_test_df.head()

decoded_df = pd.read_csv("dataset_belgium/VAE_m42v3/decoded.csv")
decoded_df = decoded_df.transpose()
decoded_df = decoded_df[1:]
decoded_df = decoded_df.transpose()
# decoded_df.head()

def trips_per_day(serie):
    l = serie.tolist()
    count = 0
    for i, value in enumerate(l):
        if l[i] == 13 and len(l) == 0:
            count += 1
        elif l[i] == 13 and l[i-1] != 13:
            count += 1
    return count

def percentage_travel_per_day(serie):
    se = serie.value_counts(normalize=True)
    if 13 in se.index:
        return se.loc[13]*100
    else:
        return 0

def n_activities_outside(serie):
```

```
#only takes into account the activity one time
    se = list(serie.unique())
    if 13 in se:
        se.remove(13)
    if 2 in se:
        se.remove(2)
    return len(se)

def n_hours_out_home(serie):
    l = serie.tolist()
    ll = [i for i in l if i != 2]
    hours = (len(ll)*2)/60
    return hours

def percentage_hours_out_home(serie):
    l = serie.tolist()
    ll = [i for i in l if i != 2]
    hours = (len(ll)*2)/60
    p = (hours/24)*100
    return p

def n_hours_at_work(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 3]
    hours = len(ll)*2/60
    return hours

def percentage_hours_at_work(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 3]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_at_home(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 2]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_work_trip(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 4]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_courses(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 5]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_eat_outside(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 6]
    hours = len(ll)*2/60
```

```
    p = (hours/24)*100
    return p

def percentage_hours_shopping(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 7]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_services(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 8]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_family_visit(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 9]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_walk_tour(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 10]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

def percentage_hours_sport_leisure(serie):
    l = serie.tolist()
    ll = [i for i in l if i == 11]
    hours = len(ll)*2/60
    p = (hours/24)*100
    return p

data = []

for index in x_test_df.index:
    d = {}

    d['n_trips_day_in'] = trips_per_day(x_test_df.iloc[index])
    d['n_trips_day_out'] = trips_per_day(decoded_df.iloc[index])

    d['percentage_travel_day_in'] =
percentage_travel_per_day(x_test_df.iloc[index])
    d['percentage_travel_day_out'] =
percentage_travel_per_day(decoded_df.iloc[index])

    d['n_activities_outside_in'] =
n_activities_outside(x_test_df.iloc[index])
    d['n_activities_outside_out'] =
n_activities_outside(decoded_df.iloc[index])

    d['percentage_hours_outside_in'] =
percentage_hours_out_home(x_test_df.iloc[index])
```

```
    d['percentage_hours_outside_out'] =
percentage_hours_out_home(decoded_df.iloc[index])

    d['percentage_hours_at_work_in'] =
percentage_hours_at_work(x_test_df.iloc[index])
    d['percentage_hours_at_work_out'] =
percentage_hours_at_work(decoded_df.iloc[index])

    d['percentage_hours_at_home_in'] =
percentage_hours_at_home(x_test_df.iloc[index])
    d['percentage_hours_at_home_out'] =
percentage_hours_at_home(decoded_df.iloc[index])

    d['percentage_hours_work_trip_in'] =
percentage_hours_work_trip(x_test_df.iloc[index])
    d['percentage_hours_work_trip_out'] =
percentage_hours_work_trip(decoded_df.iloc[index])

    d['percentage_hours_courses_in'] =
percentage_hours_courses(x_test_df.iloc[index])
    d['percentage_hours_courses_out'] =
percentage_hours_courses(decoded_df.iloc[index])

    d['percentage_hours_eat_outside_in'] =
percentage_hours_eat_outside(x_test_df.iloc[index])
    d['percentage_hours_eat_outside_out'] =
percentage_hours_eat_outside(decoded_df.iloc[index])

    d['percentage_hours_shopping_in'] =
percentage_hours_shopping(x_test_df.iloc[index])
    d['percentage_hours_shopping_out'] =
percentage_hours_shopping(decoded_df.iloc[index])

    d['percentage_hours_services_in'] =
percentage_hours_services(x_test_df.iloc[index])
    d['percentage_hours_services_out'] =
percentage_hours_services(decoded_df.iloc[index])

    d['percentage_hours_family_visit_in'] =
percentage_hours_family_visit(x_test_df.iloc[index])
    d['percentage_hours_family_visit_out'] =
percentage_hours_family_visit(decoded_df.iloc[index])

    d['percentage_hours_walk_tour_in'] =
percentage_hours_walk_tour(x_test_df.iloc[index])
    d['percentage_hours_walk_tour_out'] =
percentage_hours_walk_tour(decoded_df.iloc[index])

    d['percentage_hours_sport_leisure_in'] =
percentage_hours_sport_leisure(x_test_df.iloc[index])
    d['percentage_hours_sport_leisure_out'] =
percentage_hours_sport_leisure(decoded_df.iloc[index])


    data.append(d)


dfa = pd.DataFrame(data)
```

```
dfa['diff_n_trips_day'] = abs(dfa['n_trips_day_out'] -
dfa['n_trips_day_in'])
dfa['diff_percentage_travel_day'] =
abs(dfa['percentage_travel_day_out'] -
dfa['percentage_travel_day_in'])
dfa['diff_n_activities_outside'] = abs(dfa['n_activities_outside_out']
- dfa['n_activities_outside_in'])
dfa['diff_percentage_hours_outside'] =
abs(dfa['percentage_hours_outside_out'] -
dfa['percentage_hours_outside_in'])
dfa['diff_perc_hours_at_work'] =
abs(dfa['percentage_hours_at_work_out'] -
dfa['percentage_hours_at_work_in'])
dfa['diff_perc_hours_at_home'] =
abs(dfa['percentage_hours_at_home_out'] -
dfa['percentage_hours_at_home_in'])
dfa['diff_perc_hours_work_trip'] =
abs(dfa['percentage_hours_work_trip_out'] -
dfa['percentage_hours_work_trip_in'])
dfa['diff_perc_hours_courses'] =
abs(dfa['percentage_hours_courses_out'] -
dfa['percentage_hours_courses_in'])
dfa['diff_perc_hours_eat_outside'] =
abs(dfa['percentage_hours_eat_outside_out'] -
dfa['percentage_hours_eat_outside_in'])
dfa['diff_perc_hours_shopping'] =
abs(dfa['percentage_hours_shopping_out'] -
dfa['percentage_hours_shopping_in'])
dfa['diff_perc_hours_services'] =
abs(dfa['percentage_hours_services_out'] -
dfa['percentage_hours_services_in'])
dfa['diff_perc_hours_family_visit'] =
abs(dfa['percentage_hours_family_visit_out'] -
dfa['percentage_hours_family_visit_in'])
dfa['diff_perc_hours_walk_tour'] =
abs(dfa['percentage_hours_walk_tour_out'] -
dfa['percentage_hours_walk_tour_in'])
dfa['diff_perc_hours_sport_leisure'] =
abs(dfa['percentage_hours_sport_leisure_out'] -
dfa['percentage_hours_sport_leisure_in'])

dfa.to_csv("dataset_belgium/VAE_m42v3/analysis.csv")
```

Code for the Encoder-Decoder LSTM model

```
# LSTM not binary
#all data
#only from 10:00 to 21:00

import pandas as pd
from datetime import timedelta
import tensorflow as tf
import numpy as np
from numpy import array, concatenate
from keras.models import Sequential, Model
from keras.layers import Input, LSTM, Dense

# load the data
df = pd.read_csv("dataset_belgium/merged.csv", sep=';')

df_wkd = df.loc[(df.weekday > 1.0) & (df.weekday < 7.0)]
df_yg = df_wkd.loc[(df.age < 65.0)]
del df_yg['g_regi']
del df_yg['age']
del df_yg['mainmode']
del df_yg['t_withchild']
del df_yg['weekday']
del df_yg['h_hierarchie_urbaine']

df_yg = df_yg.transpose()

df_yg = df_yg[1:]
timeline = []
for timestep in range(0,len(df_yg.index)):
    time = timedelta(seconds=timestep*2*60)
    time = str(time)
    timeline.append(time)

td = pd.TimedeltaIndex(timeline, unit = 'm', freq = 'infer', name =
'timeline')
df_yg = df_yg.set_index(td)

#take only timesteps multiple of 4 to make calculations easier after
timestep_6 = []
for i in range (0,720,3):
    timestep_6.append(i)
df_10 = df_yg.iloc[timestep_6]
df = df_10.transpose()

#from pandas dataframe to array of rows
L = []
for index in df.index:
    #only from 10:00 to 21:00
    L.append(df.loc[index][100:210])

dta = array(L)
X_ini = np.expand_dims(dta, axis=2)

#get sequences of n_steps_in timesteps and predict the next
n_steps_out timesteps
```

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
    #X is input sequence, y is output sequence, y_1 is output sequence
one timestep forward
    #the last n_steps_in + n_steps_out are omitted because they would
be out of range (problem?)
    X, y, y_1 = list(), list(), list()
    for i in range(len(sequence)):
        # find the end of this pattern
        """check"""
        end_ix = i + n_steps_in
        # check if we are beyond the sequence
        if end_ix > len(sequence)-1:
            break
        # gather input and output parts of the pattern
        seq_x = sequence[i:end_ix]
        seq_y_1 = sequence [end_ix:end_ix+n_steps_out]
        seq_y = np.concatenate((array([[0]]),seq_y_1[:-1]))
        if len(seq_y) == n_steps_out and len(seq_y_1) == n_steps_out:
            X.append(seq_x)
            y.append(seq_y)
            y_1.append(seq_y_1)
    return array(X), array(y), array(y_1)

# choose a number of time steps
n_steps_in = 70 #from 10:00 to 17:00
n_steps_out = 40 #from 17:00 to 21:00

#prepare the data
encoder_input_data, decoder_input_data, decoder_target_data =
array([]), array([]), array([])

number_of_samples_to_train_on = 6000

for i, seq in enumerate(X_ini[:number_of_samples_to_train_on]):
    x, y, y_1 = split_sequence (seq, n_steps_in, n_steps_out)
    if i == 0:
        encoder_input_data = x
        decoder_input_data = y
        decoder_target_data = y_1
    else:
        encoder_input_data = np.append(encoder_input_data, x, axis=0)
        decoder_input_data = np.append(decoder_input_data, y, axis=0)
        decoder_target_data = np.append(decoder_target_data, y_1,
axis=0)

X_flat = encoder_input_data.reshape(-1)
y_flat = decoder_input_data.reshape(-1)
y_1_flat = decoder_target_data.reshape(-1)

nb_classes = 14

def indices_to_one_hot(data, nb_classes):
    return np.eye(nb_classes)[data]

#one-hot encode variables
xt1 = indices_to_one_hot(X_flat, nb_classes)
```

```
X =
xt1.reshape(encoder_input_data.shape[0],encoder_input_data.shape[1],nb
_classes)

yt2 = indices_to_one_hot(y_flat, nb_classes)
y =
yt2.reshape(decoder_input_data.shape[0],decoder_input_data.shape[1],nb
_classes)

yt2_1 = indices_to_one_hot(y_1_flat, nb_classes)
y_1 =
yt2_1.reshape(decoder_target_data.shape[0],decoder_target_data.shape[1
],nb_classes)

# returns train, inference_encoder and inference_decoder models
def define_models(n_input, n_output, n_units):
    #n_input = number of features of input sequence
    #n_output = number of features of output sequence
    #n_units is the dimension of the latent space

    # define training encoder
    encoder_inputs = Input(shape=(None, n_input))
    encoder = LSTM (n_units, return_state = True)
    encoder_outputs, state_h, state_c = encoder(encoder_inputs)
    encoder_states = [state_h, state_c]
    #define training decoder
    decoder_inputs = Input(shape=(None, n_output))
    decoder_lstm = LSTM(n_units, return_sequences = True,
return_state=True)
    decoder_outputs, _, _= decoder_lstm(decoder_inputs,
initial_state=encoder_states)
    decoder_dense = Dense(n_output, activation='softmax')
    decoder_outputs = decoder_dense(decoder_outputs)
    model= Model([encoder_inputs, decoder_inputs], decoder_outputs)
    #define inference encoder
    encoder_model = Model(encoder_inputs,encoder_states)
    #define inference decoder
    decoder_state_input_h = Input(shape=(n_units,))
    decoder_state_input_c = Input(shape=(n_units,))
    decoder_states_inputs = [decoder_state_input_h,
decoder_state_input_c]
    decoder_outputs, state_h, state_c = decoder_lstm(decoder_inputs,
initial_state=decoder_states_inputs)
    decoder_states = [state_h, state_c]
    decoder_outputs = decoder_dense(decoder_outputs)
    decoder_model = Model([decoder_inputs] + decoder_states_inputs,
[decoder_outputs] + decoder_states)
    #return all models
    return model, encoder_model, decoder_model

#generate target given source sequence
def predict_sequence(infenc, infdec, source, n_steps, cardinality):
    #encode
    state = infenc.predict(source)
    #start of sequence input
    target_seq = np.array([0 for _ in range(cardinality)]).reshape(1,
1, cardinality)
    #collect predictions
    output = list()
```

```
    for t in range(n_steps):
        #predict next char
        yhat, h, c = infdec.predict([target_seq] + state)
        #store prediction
        output.append(yhat[0,0,:])
        #update state
        state = [h,c]
        #update target sequence
        target_seq = yhat
    return np.array(output)

# configure problem
n_features = 14
latent_dim = 128

# define model
train, infenc, infdec = define_models(n_features, n_features,
latent_dim)
train.compile(optimizer='rmsprop', loss='categorical_crossentropy',
metrics=['accuracy'])

# train model
train.fit([X, y], y_1, batch_size=32, epochs=10, validation_split=0.2

time_in = []
for column in df.columns[100:170]:
    time_in.append(column)
t_in = pd.TimedeltaIndex(time_in, unit = 'm', freq = 'infer', name =
'timeline')
t_in = t_in.to_frame()

time_out = []
for column in df.columns[170:210]:
    time_out.append(column)
t_out = pd.TimedeltaIndex(time_out, unit = 'm', freq = 'infer', name =
'timeline')
t_out = t_out.to_frame()

input_df = t_in.transpose()
output_df = t_out.transpose()
predicted_df = t_out.transpose()

input_df = input_df[1:]
output_df = output_df[1:]
predicted_df = predicted_df[1:]

for i in range(0,873):
    #get one sample of the dataset
    sample_number = number_of_samples_to_train_on+i
    n_steps_in = 70
    n_steps_out = 40
    x_pred, y_pred, y_1_pred = split_sequence (X_ini[sample_number],
n_steps_in, n_steps_out)

    #one-hot encode the sample
    x_pred_flat = x_pred.reshape(-1)
    xt = indices_to_one_hot(x_pred_flat, nb_classes)
    X_pred = xt.reshape(x_pred.shape[0], x_pred.shape[1], nb_classes)
```

```
    #make a prediction (inference)
    target = predict_sequence(infenc, infdec, X_pred, n_steps_out,
n_features)

    #one-hot decode sequences
    decoded_target = np.argmax(target, axis=1)
    input_sentence = np.argmax(X_pred, axis=2)

    input_sentence = list(input_sentence.flatten())
    y_1_pred = list(y_1_pred.flatten())
    decoded_target = list(decoded_target.flatten())

    input_df.loc[i] = input_sentence
    output_df.loc[i] = list(y_1_pred)
    predicted_df.loc[i] = list(decoded_target)

input_df.to_csv("dataset_belgium/LSTM/input_seqs.csv")
output_df.to_csv("dataset_belgium/LSTM/output_seqs.csv")
predicted_df.to_csv("dataset_belgium/LSTM/predicted_seqs.csv")
```