Résumé

L'augmentation du trafic aérien induit la formation de zones de haute complexité dans certains secteurs. Ces "points chauds" ou "bouffées de trafic" nécessitent une surveillance accrue de la part des contrôleurs aériens, afin de décider si des actions sont nécessaires sur certaines trajectoires pour éviter un conflit. Réduire cette charge de travail accrue serait possible si la survenue de ces points chauds pouvait être prédite plusieurs dizaines de minutes à l'avance et des actions proposées avant leur formation. Dans ce rapport, nous proposons plusieurs approches visant à prédire les zones de congestion à une telle échelle de temps à l'aide de techniques issues du machine learning et plus précisémment des réseaux de neurones récurrents (RNN). Les réseaux de neurones récurrents sont particulièrement adaptés pour traiter des séries temporelles, et donc notamment des trajectoires d'avions. Les zones de congestion sont définies à l'aide de métriques de complexité prenant en compte la structure intrinsèque des trajectoires. Divers modèles s'appuyant sur les réseaux de neurones récurrents sont testés, en particulier en s'inspirant des architectures encodeur-décodeur utilisées dans les tâches de traduction automatique. Un modèle de prédiction de trajectoire s'appuyant également sur les réseaux de neurones récurrents est aussi proposé.

Mots clés : gestion du trafic aérien, métriques de complexité, réseaux de neurones récurrents, prédiction de trajectoire

Abstract

The increase of air traffic induces the formation of high complexity areas in some sectors of the airspace. These "hot spots" or "traffic bursts" require an intense surveillance from air traffic controllers in order to decide if some trajectories should be modified to avoid any conflict. Reducing this increased workload would be possible if the occurrence of these hot spots could be predicted tens of minutes before their formation, and if actions could be suggested in advance. In this report, we propose several approaches to predict congested areas at a such time scale relying on machine learning techniques and more precisely on recurrent neural networks (RNN). These recurrent neural networks are well suited to deal with time series, and in particular aircraft trajectories. The congested areas are defined using complexity metrics taking into account the intrinsic structure of trajectories. Various models relying on recurrent neural networks are tested, in particular using encoder-decoder architectures drawn from automatic translation tasks. A trajectory prediction model using recurrent neural networks is also proposed.

Keywords: air traffic management, complexity metrics, recurrent neural networks, trajectory prediction

Acknowledgements

This internship, taking place at the ENAC laboratory, is made possible thanks to the supervision of Professor Daniel Delahaye. I would also like to thank Professor Nidhal Bouaynaya and Professor Ghulam Rasool from the Rowan University, as well as Ian Levitt, Andrew Cheng and Albert Schwartz from the FAA WJH Technical Center and Elsa Birman from the CRNA-EST for helping me with their respective expertise in Natural Language Processing and Air Traffic Flow Management.

Table of contents

Table of figures

Chapter One

Introduction

With the continuous rise in air transportation demand (a 4.3% annual growth of the worldwide passenger demand is expected until 2035[1]), the capacity of the Air Traffic Management (ATM) system reaches its limits, leading to increased flight delays (expected to achieve 8.5 minutes per flight in 2035 with the current system[2]). The Covid-19 crisis has severely impacted the aviation industry, with a decreased down to 90% of RPK (Revenue Passenger Kilometer) in July 2020 with respect to the same period in 2019 according to IATA[3]. However, the traffic is expected to regain its growth rate once the crisis is over, with the worst-case scenario predicting a return to normalcy before 2025. From the air traffic controllers perspective, potential converging proximity are strongly demanding, because they have to constantly monitor numerous trajectories to decide if they should act on them or not. In particular, some sectors of the airspace are known to develop high complexity areas, also referred as "hot spots" or "traffic bursts". Even if the Flow Management Position (FMP) is able to balance the capacity with the forecast demand, the scale of the predictions is too crude to avoid the formation of hot spots between the end of the strategic planning (about one hour before the entry in the sector) and the controller's tactical level.

The subject of this internship is: *Predicting congested areas at tactical time with recurrent neural networks*. In a time horizon lower than 20 minutes, existing Medium Term Conflict Detection (MTCD) tools provide efficient conflicts detection. The objective of this work is to apply machine learning methods to provide congestion predictions in an extended time horizon, between 40 to 60 minutes. For this time horizon, the uncertainties on the trajectories make it difficult to predict the exact trajectories that will be involved in a conflict. This is why we will focus on predicting congested areas rather than individual trajectories involved in conflicts. Congestion appears when a set of trajectories strongly interact in a given area and time interval, leading to potential conflicts and hence requiring high monitoring from the controllers. The aim of predicting congested areas is to provide a decision making tool for the FMP, providing information

[1]ICAO. Long-Term Traffic Forecasts, Passenger and Cargo, 2018
[2]SESAR Joint Undertaking. Airspace Architecture Study, 2019
[3]`https://www.iata.org/en/pressroom/pr/2020-09-29-02/`

to help taking actions at tactical time (less than one hour) such as rerouting or sector configurations. The machine learning advances over the last decades have already been successfully applied in various problems of Air Traffic Management, for example to predict estimated take-off times [7], or to detect controllers' actions [33]. In this application, the objective is to make predictions by using sequences of aircraft states (trajectories). Recurrent Neural Networks (RNN) are well suited to deal with sequences of data, and this is why they have been chosen to build prediction models for congested areas. In order to quantify the notion of congestion and to design a supervised learning task, a measure of the air traffic complexity is required. This will be achieved by using a complexity metric already developed in the literature.

The chapter two provides the state of the art. Four brief literature reviews are proposed. The first one presents the related works on complexity metrics. In the second one, we introduce the basic concepts of Recurrent Neural Networks, while some well known deep learning techniques are outlined in the third review. Finally, a short literature review on Trajectory Prediction with Machine Learning approaches is conducted. In section 2.5, two software previously developed at the ENAC laboratory and used throughout this project are presented, along with a brief description of the datasets coming from external third-parties. Chapters three, four, and five present three different approaches developed during this internship to directly predict congested areas, along with their respective results. Chapter three implements a simple RNN model trained on a regression task aiming at predicting the complexity value (computed with the complexity metric) at every point of the airspace in 40 minutes. Chapter four addresses the same task with more sophisticated models drawn from the Natural Language Processing field. Chapter five proposes a different definition of congested areas which are seen as "object" to be detected in the past and current trajectories. Chapter six is a first sketch of an indirect approach where RNN are used for Trajectory Prediction (TP) before computing the complexity. Finally, chapter seven suggests several options to design a more efficient model and to evaluate it, as well as new applications that could benefit from it.

The code developed during this internship can be found on my github page [4].

[4] `https://github.com/lshigarrier/PFE`

Chapter Two

State of the art

The objective of this chapter is twofold. First, a literature review reflecting on four domains treated in this work is proposed. Section 2.1 presents a sample of the state-of-the-art complexity metrics for Air Traffic Management with an emphasis on the metric retained for the rest of this report. Section 2.2 is a brief introduction to recurrent neural networks. Some deep learning techniques required for the understanding of the models presented in the following chapters are outlined in section 2.3. Then, a review of Trajectory Prediction using Machine Learning approaches is provided in section 2.4.

Secondly, we introduce the software and datasets used as inputs for this project. Section 2.5 presents the software used to compute the complexity metric selected in section 2.1 as well as the Aircraft Arithmetic Simulator used to generate trajectories for the Trajectory Prediction models (presented in the chapter six). Both these software have been previously developed at the ENAC laboratory. Finally, the sources of the others datasets used during this internship are presented.

2.1 Complexity metrics

In this section, we will briefly review the main approaches to complexity metrics developed in the literature. Extensive reviews on this topic can be found in [38]. The literature focusing on conflict detection and estimation of conflict probability (see [29] for one example) is not addressed in this review.

The traditional measure of the air traffic complexity is the traffic density, defined as the number of aircraft crossing a given sector in a given period [19]. The traffic density is compared with the operational capacity, which is the acceptable number of aircraft allowed to cross the sector in the same time period. This crude metric does not take into account the traffic structure and the geometry of the airspace. Hence, a controller may continue to accept traffic beyond the operational capacity, as well as refusing aircraft even though the operational capacity has not been reached.

The Dynamic Density (introduced in [24]) aims at producing an aggregate measure of

complexity by combining complexity factors, including static air traffic characteristics (such as airway crossings) and dynamic air traffic characteristics (such as the number of aircraft, the closing rates etc.). A list of these complexity factors can be found in [19]. The complexity factors can be combined either linearly (for example [43]) or through a neural network ([4], [5]). Several versions of Dynamic Density have been proposed in the literature ([23], [30]). For example, the interval complexity (introduced in [12]) is a variant of Dynamic Density where the chosen complexity factors are averaged over a time window.

Another approach is the Fractal Dimension (introduced in [31]). Fractal Dimension is an aggregate metric for measuring the geometrical complexity of a traffic pattern which evaluates the number of degrees of freedom used in a given airspace. The fractal dimension is a ratio providing a statistical index of complexity comparing how detail in a pattern changes with the scale at which it is measured. Applied to air route analysis, it consists in computing the fractal dimension of the geometrical figure composed of existing air routes. A relation between fractal dimension and conflict rate (number of conflicts per hour for a given aircraft) is also shown in [31].

In [26], an Input-Output approach to traffic complexity is exposed. In this approach, air traffic complexity is defined in terms of the control effort needed to avoid the occurrence of conflicts when an additional aircraft enters the traffic. The input-output system is defined as follow. The air traffic within the considered region of the airspace is the system to be controlled. The feedback controller is an automatic conflict solver. The input to the closed-loop system is a fictitious additional aircraft entering the traffic. The output is computed using the deviations of the aircraft already present in the traffic due to the new aircraft (issued by the automatic conflict solver). This amount of deviations needed to solve all the conflicts provides a measure of the air traffic complexity. This metric is dependent on the conflict solver and on the measure of the control effort.

Another approach of complexity metrics is the intrinsic complexity metrics approach (for example in [21]). In this case, a difference is made between the control workload and the traffic complexity [32]. The control workload is a measurement of the difficulty for the traffic control system (human or not) of treating a situation. This is linked to the cognitive process of traffic situation management. The traffic complexity, on the other hand, is an intrinsic measurement of the complexity of a traffic situation, independently to any traffic control system. Intrinsic complexity metrics ignore the control workload and aim at modelling the traffic complexity only using the geometry of trajectories. In [9], an intrinsic complexity metrics is described using linear and nonlinear dynamical system models. The air traffic situation is modeled by an evolution equation (the aircraft trajectories being integral lines of the dynamical system). The complexity is measured by the Lyapunov exponents of the dynamical system. Intuitively, The Lyapunov exponents capture the sensitivity of the dynamical system to initial conditions. In this study, we will use a similar approach, based on linear dynamical system computed in the neighborhood of a trajectory sample and taking into account the uncertainty on the position of the considered aircraft. With this metric, a linear dynamical system is fitted to the aircraft speed vectors V and position vectors X, such that $V \simeq A. \cdot X + b$. The eigenvalues of A are used to identify different organizational structures of the aircraft speed vectors, such

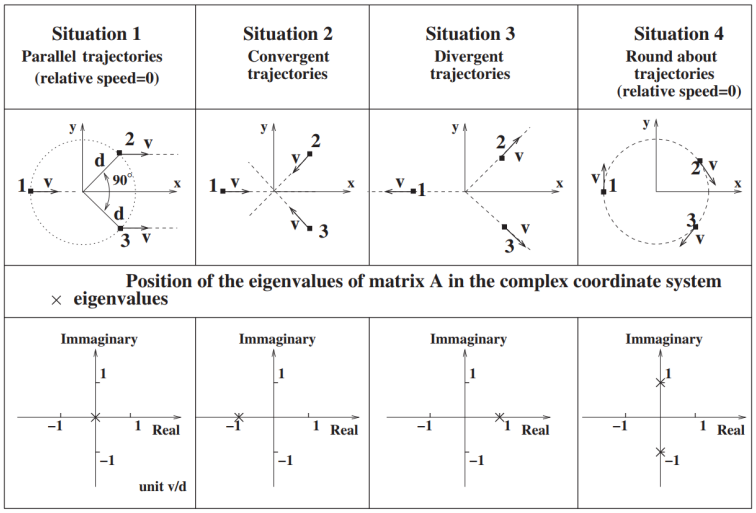


Figure 2.1: Eigenvalues loci for several typical situations.The small squares are the initial positions of aircraft at a given time (this represents the observation given by a radar for instance with the associated speed vector). Source: [9].

as translation, rotation, divergence, convergence, or a mix of them (see Figure 2.1 taken from [9]). This last complexity metric is computationally efficient [14].

A Dynamic Weighted Network Approach is introduced in [48]. Three types of complexity relations are defined: aircraft-aircraft (risk of conflicts), aircraft-waypoints (i.e the proximity with the entry/exit points of a sector) and aircraft-airways (deviation of an aircraft from its route). These complexity relations are combined in a dynamic network, where the nodes are the air traffic units (aircraft, waypoints and airways) and the edges are the complexity relations between them, weighted by a measure of this complexity. A aggregate complexity metric is then derived from this network.

In [49], the authors define flight conflict shape movements on each point of an aircraft's trajectory based on the aircraft position and speed, as well as its likely angle of deviation and the safety standards of separation. Using this flight conflict shape movement, a complexity value is attributed to each pair of aircraft. Then, an average measure of complexity can be computed for a user-defined area. The complexity metric thus defined is compared with the intrinsic metric from [9]. Despite its simplicity, this complexity metrics is able to account for the complexity in typical scenarios.

2.2 Introduction to Recurrent Neural Networks

In this section, we will present a brief overview of recurrent neural networks (RNN). The basic RNN layer is detailed first, along with a typology of the tasks that are efficiently achieved by a RNN. Then, we will describe some RNN variants such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU). Some Natural Language Processing (NLP) concepts including beam search algorithm and attention mechanisms are briefly

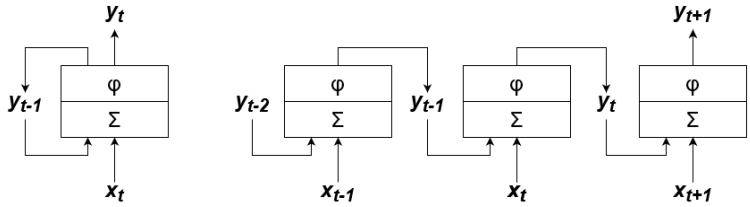


Figure 2.2: Basic RNN layer (left) and the same layer unrolled through time (right). Σ represents linear mapping and ϕ represents the activation function

detailed in the next section (2.3). A complete review of other NLP concepts (language models, word embedding etc.) can be found in [50].

RNN is a class of nets that is designed to process sequences of data rather than fixed-sized inputs. Hence, they are very effective at analysing and predicting time series, and have been extensively used in Natural Language Processing including automatic translation (for example [44]) or speech-to-text (for example [41]). They have also been applied in generative models, for example in image captioning ([47]).

The basic RNN layer is shown in Figure 2.2. At each time step t, the recurrent cell receives the input x_t as well as its own output from the previous time step y_{t-1}. The output y_t is computed according to Equation (2.1), where W is the weights matrix, b is the bias vector and ϕ is an activation function (for example *tanh* or ReLU, [25]). x_t and y_{t-1} have been concatenated in a single row vector denoted $[x_t; y_{t-1}]$.

$$y_t = \phi(W \cdot [x_t; y_{t-1}] + b) \tag{2.1}$$

Depending on the task aimed to be solved by the network, various architectures are possible. A simple typology of these architectures is provided in Figure 2.3 taken from [22]. One-to-one networks correspond to classical (non-recurrent) feedforward networks. In one-to-many networks, a single input vector is used to output a sequence, for example in image captioning. Alternatively, in many-to-one networks, a sequence of inputs can be used to get a single output vector, for example in sentiment analysis. The first many-to-many architecture is an *encoder-decoder*, where a sequence of inputs is encoded into a single vector, then decoded into a sequence of outputs. This type of model can be used for automatic translation, where we need to read the full input sequence before beginning the translation process. More details concerning encoder-decoder model for automatic translation are provided in section 2.3. Finally, the second many-to-many model also transforms a sequences of inputs into a sequences of outputs, but a new output is generated right after a new input is provided to the network. An example of application is the video classification, where each frame of a video is labeled.

To train the network, the classical optimization algorithms (generally based on gradient descent) need the gradients of the parameters with respect to the outputs. To compute these gradients, we can simply use the backpropagation method, often referred as *backpropagation through time* (BPTT) when applied to RNN.

To prevent overfitting, a technique often used with RNN is dropout ([36], [13]). When

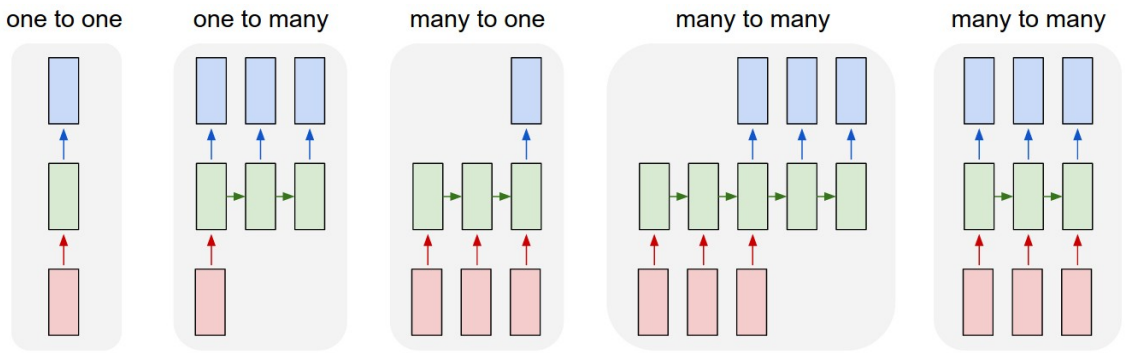


Figure 2.3: Each rectangle is a vector and arrows represent functions (e.g. matrix multiply). Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state. In the basic RNN layer, the RNN's state is equal to the output vector, but they are generally different in more complex layers (i.e LSTM). Source: [22].

applied to a given layer, dropout consists in randomly remove (set to zero) a fixed proportion of the outputs of the layer. The cancelled outputs changed at each forward pass in the network. The dropout proportion (also referred as the dropout rate) is an hyperparameter.

The basic RNN layer introduced above has difficulty learning long term dependencies when the length of the sequence is very long. This problem has been referred as the *vanishing gradient* problem [35]. To solve it, several architectures of RNN layer with long-term memory have been introduced. We will present two of these new type of layers: Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU).

Long Short-Term Memory were first introduced in [20], and then improved over the years (for example [40] and [51]). LSTM were designed to be faster to train than regular RNN, and to be able to detect long-term dependencies. Figure 2.4 shows the architecture of the LSTM cell. Contrary to the basic RNN layer, the LSTM has two state vectors denoted h_t (the hidden state) and c_t (the cell state) which can be seen respectively as the short-term state and the long-term state. The LSTM's equations are presented in Equations (2.2) to (2.7). Symbol $\cdot$ represents the matrix multiplication, while symbol $\times$ represents the element-wise multiplication. σ is the logistic activation (to get outputs between 0 and 1). The current input x_t and the previous hidden state h_{t-1} are used to compute the controllers of three *gates*: the input gate controller i_t, the forget gate controller f_t and the output gate controller o_t (Equations (2.2) to (2.4)). These gate controllers are simply computed by a linear mapping followed by a logistic activation. In parallel, a candidate cell state $\tilde{c}_t$ is computed using the current input and the previous hidden state (Equation (2.5)). Equation (2.5) can be seen as the LSTM equivalent of the equation of the basic RNN layer (Equation (2.1)). In Equation (2.6), the current cell state c_t is then updated by adding the previous cell state c_t (filtered by the forget gate) and the candidate cell state $\tilde{c}_t$ (filtered by the input gate). The hidden state h_t is finally computed by activating the cell state c_t with the *tanh* function (or another activation function such as the ReLU), which is then passed though the output gate (Equation (2.7)). With the input gate i_t,

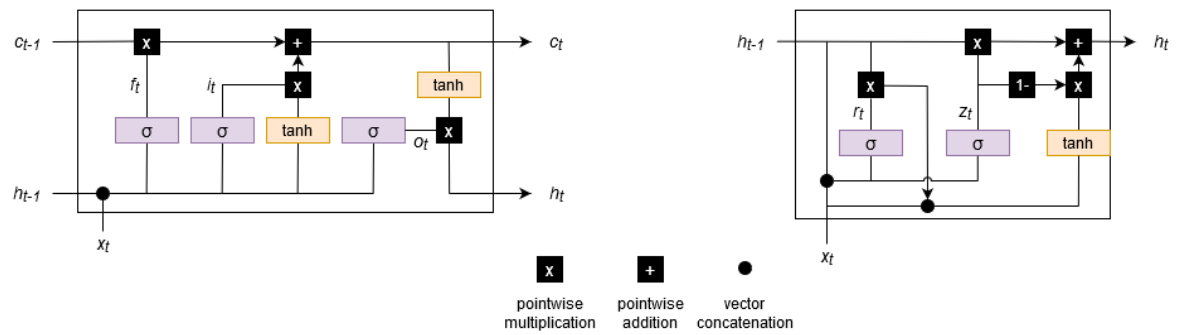


Figure 2.4: LSTM cell (left) and GRU cell (right)

the LSTM is able to store information in the cell state c_t, to extract it with the output gate o_t and to discard it with the forget gate f_t.

$$i_t = \sigma(W_i \cdot [x_t; h_{t-1}] + b_i) \tag{2.2}$$
$$f_t = \sigma(W_f \cdot [x_t; h_{t-1}] + b_f) \tag{2.3}$$
$$o_t = \sigma(W_o \cdot [x_t; h_{t-1}] + b_o) \tag{2.4}$$
$$\tilde{c}_t = tanh(W_c \cdot [x_t; h_{t-1}] + b_c) \tag{2.5}$$
$$c_t = f_t \times c_{t-1} + i_t \times \tilde{c}_t \tag{2.6}$$
$$h_t = o_t \times tanh(c_t) \tag{2.7}$$

A variant of LSTM (introduced in [15]) called *peephole connections* uses the cell states c_{t-1} and c_t to compute the gate controllers. Another variant of the LSTM is the Gated Recurrent Unit (GRU) introduced in [6]. Figure 2.4 illustrates the GRU cell and its equations are detailed in Equations (2.8) to (2.11). The GRU is a simplified version of the LSTM, but seems to have the same performance (see [17]). The GRU has a unique state vector h_t, while the forget and input gates are merged into a single gate controller z_t (Equation (2.8) and (2.11)). The output gate is replaced by another gate controller r_t (Equation (2.9)), placed between the previous hidden state h_{t-1} and the candidate hidden state $\tilde{h}_t$ (Equation (2.10)).

$$z_t = \sigma(W_z \cdot [x_t; h_{t-1}] + b_z) \tag{2.8}$$
$$r_t = \sigma(W_r \cdot [x_t; h_{t-1}] + b_r) \tag{2.9}$$
$$\tilde{h}_t = tanh(W_h \cdot [x_t; r_t \times h_{t-1}] + b_h) \tag{2.10}$$
$$h_t = z_t \times h_{t-1} + (1 - z_t) \times \tilde{h}_t \tag{2.11}$$

An improvement of the RNN structure is the Bidirectional Recurrent Neural Network (BRNN) introduced in [42]. In the basic RNN layer, the hidden state h_t is computed using only the sequence of inputs $x_0, ..., x_t$. However, the prediction at time step t may depend on elements that are further in the input sequence (for example in automatic

translation). BRNN consists in creating two networks, with one of these networks going through the input sequence in a forward pass (x_0 until x_T where T is the length of the input sequence), and the other one going through the input sequence backward (x_T until x_0). Hence, we obtain two sequences of hidden states, and at each time step, the prediction can be computed using these two hidden states. The main drawback of BRNN is that the full input sequence is needed to produce predictions, so online predictions is impossible.

2.3 Some elements of Deep Learning

In this section, we will focus on three elements of deep learning that are used in the applications developed in the remaining chapters of this report. We will first provide a general insight of two model's architectures : the encoder-decoder architecture used for automatic translation, and the YOLO architecture used for object recognition. Then, an overview of the classical optimizers used in deep learning is presented with an emphasis on the Adam algorithm which has been used in all the experiments conducted during this work.

Encoder-decoder model

In this paragraph, we describe the encoder-decoder network architecture that is then used in chapters four, five and six. The first introduction of an encoder-decoder model for automatic translation is provided in [44]. This model has also been applied to various tasks, such as speech recognition [37] or video captioning [46].

Encoder-decoder models are used to map a fixed-length input sequence to a fixed-length output sequence where the length of the input differs from the length of the output. The basic principles of an encoder-decoder model are shown in the Figure 2.5. The model is composed of two parts: an encoder network and a decoder network. The encoder network is made of recurrent layers (such as LSTM or GRU layers presented in the previous section). It receives in input the input sequence and encodes it into one or more encoding vectors, define as the hidden states computed during the last pass through the recurrent layers. The decoder network is also made of recurrent layers. The hidden states of these layers are initialized with the encoding vectors computed by the encoder network. The sequence used as input for the recurrent layers of the decoder network is dependent on the ongoing process: training or inference. This point is detailed below. Either way, given the initial hidden states and a suitable input sequence, the decoder network is able to compute an output sequence with a length different from the input sequence's length.

An important aspect of this architecture is the difference between the training procedure (when the parameters of the network are optimized) and the inference procedure (when an output sequence is generated by the network given an input sequence). The training procedure is straightforward. A pair of input and output sequences is provided. The

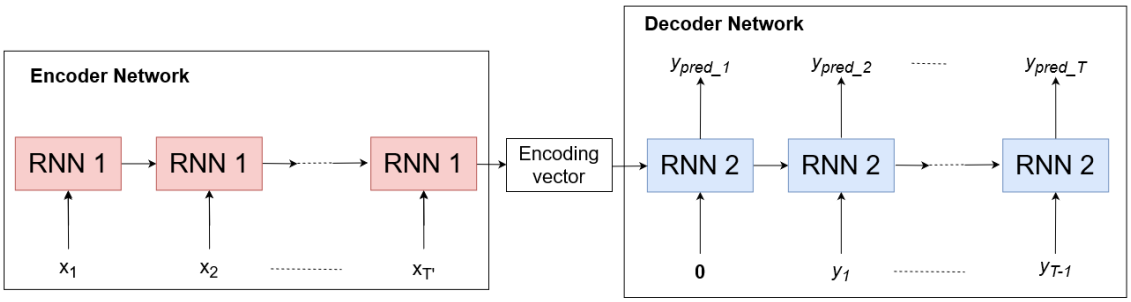


Figure 2.5: Basic architecture of an encoder-decoder model

input of the decoder network is defined as the true output sequence shifted from one timestamp to the right, such that the decoder network receives as input the vector it should have predicted at the previous timestamp. The first element of the input of the decoder network is simply a zero vector. An error (for example the mean squared error) can then be computed between the predicted sequence and the true output sequence, and be backpropagated to compute the gradients of the error with regards to each parameter of the network.

During the inference procedure, the true output sequence is obviously not known, hence the input of the decoder network has to be redefined. A simple method consists in recursively using the prediction of the last timestamp as the input for the next one. However, this method will certainly accumulate prediction errors and lead to poor performances if the output sequence is long enough. A solution to this issue is to change the task of the decoder network, such that the network do not predict an element of the output sequence at each timestamp, but rather a distribution over the possible outputs. Then, using a beam search algorithm, it is possible to maintain at each timestamp a small number of possible output sequences using the partial likelihood of each sequence as the metric. When the last timestamp is reached, the model can output the sequence with the highest likelihood.

We illustrate the beam search algorithm with an example. Let e and d be respectively the trained encoder and the trained decoder. The encoder e is a function that takes an input sequence and returns an encoding vector v. The decoder d is a function that takes the encoding vector v as well as the beginning of an output sequence $y_1, \dots, y_t$ and returns the distribution $P(y_{t+1}|v, y_1, \dots, y_t)$ of the next element of the output sequence y_{t+1} given the encoding vector and the first terms of the output sequence. Let $x_1, \dots, x_{T'}$ be an input sequence. For example, $x_1, \dots, x_{T'}$ can be seen as a sentence where each x_i is the representation of a word obtained with word embedding (see [50] for more details about word embedding). We suppose in this example that the output sequences $y_1, \dots, y_T$ are built with elements y_t taken from a finite dictionary D. In the case of the automatic translation task, each y_t is a word from the target language. Our final objective would be to compute the distribution $P(y_1, \dots, y_T|x_1, \dots, x_{T'})$ and then take the argmax to get the output sequence $y_1, \dots, y_T$ maximizing this distribution given the input sequence. This distribution is unfortunately impossible to compute in a reasonable amount of time, this is why an heuristic is used, namely the beam search algorithm. The first step is to

compute the encoding vector $v = e(x_1, \ldots, x_{T'})$. Then, the distribution $P(y_1|v) = d(v)$ is computed, using a zero vector as input of the decoder network. We introduce here a parameter B of the beam search corresponding to the number of sequences that will be kept at each timestamp. So, using the distribution $P(y_1|v)$, we store the B sequence starts with the highest partial likelihood $y_1^1, \ldots, y_1^B$. Then, each of these sequence starts are used as inputs of the decoder: for every j from 1 to B, we compute the distribution $P(y_2|v, y_1^j)$. For every j from 1 to B and for every element y_2 in the dictionary D, we can compute the partial likelihood $P(y_1^j, y_2|v) = P(y_2|v, y_1) \times P(y_1^j|v)$. Once again, we only keep the B partial sequence y_1, y_2 with the highest partial likelihood $P(y_1^j, y_2|v)$. By repeating this process for each timestamp, we end up with B complete output sequences $y_1, \ldots, y_T$. It is then possible to output the sequence with the highest likelihood among these B sequences. Note there is no guarantee that this is the sequence that maximize $P(y_1, \ldots, y_T|v)$.

The full potential of this basic framework is only achieved when it is combined with several improvements. For example, reversing the order of the input sequence has been shown to significantly improve the performance of the model ([44]). Another family of improvements used in many deep learning models is attention mechanisms. The idea behind attention mechanisms is that, at each timestamp of the decoding process, the decoder should focus on specific parts of the input sequence, and not on the whole input sequence as encoded into the encoding vector. To achieve this goal, the decoder should be able to access all the hidden states of the encoder network (one for each element of the input sequence) rather than only access the last hidden state, namely the encoding vector. Several architectures are possible to implement an attention mechanism (for example [2], [28]). The general idea is to train a multilayer perceptron taking as input the last decoder hidden state as well as all the encoder hidden states, and outputting a vector of weights which sum to 1 (using a softmax activation), such that each hidden state of the encoder is associated with a weight. It is then possible to sum all the hidden states of the encoder weighted by the output of the multilayer perceptron, resulting in a vector called the context vector. This context vector along with the last output of the decoder can been used as the input for the next timestamp of the decoder.

What has been described here is in fact a special case of attention-based mechanisms called General Attention. Another type of attention mechanisms are Self-Attention, which apply attention only among the inputs. This type of attention mechanisms is notably used in the Transformers architecture (introduced in [45]) which is currently the state-of-the-art architecture for most NLP tasks.

YOLO algorithm for object detection

In this paragraph, we sketch a short presentation of the YOLO algorithm introduced in [39] to address the object detection task. We take a loose inspiration from this approach for the model developed in chapter five aiming at detecting congested areas defined as clusters with a specific location, extension and orientation. Object detection has to be distinguished from object localization. In object localization, an image is fed into the model and the output is a class, for example the image is classified as a 'plane' or a 'car'

etc, along with the position of the object in the image. However, object localization do not provide any information on the number of such objects, which could possibly fall into different classes. On the contrary, the object detection task aims at providing the locations, shapes, numbers and classes of all the objects (possible none) of the image. An immediate application of object detection algorithm is the field of self-driving cars, where an algorithm has to detect the positions of various classes of objects (cars, pedestrians, traffic signs etc.) within a time constraint.

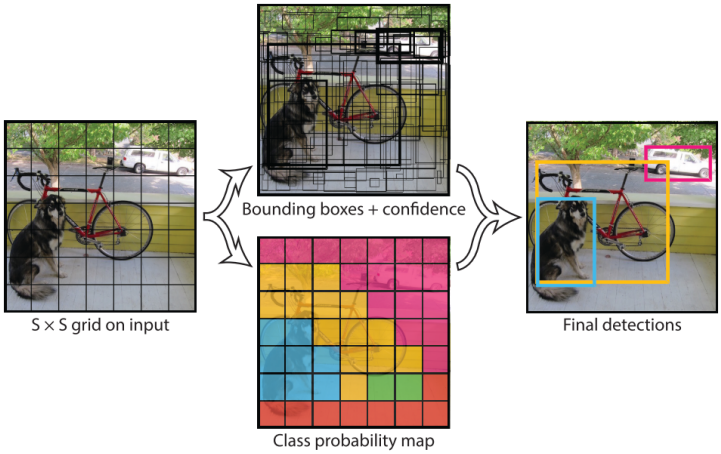


Figure 2.6: Model of the object detection task from [39]

Several variants of YOLO have been developed during the last years. In this brief introduction, we will only provide one simple approach. YOLO is an acronym for "You Only Look Once". Former object detection models were based on classification models. To detect objects, the image has to be fed several times into a classification model, each time with a different cropping using sliding windows covering the entire image. The idea of the YOLO algorithm is to fed the image only one time into the model, hence significantly improving the time performances of the object detection task, while learning a general representation of the image (rather than a local one with the sliding windows approach). To achieve this goal, each image is divided into a $N \times N$ grid (see Figure 2.6). For each grid cell, a bounding box is defined. A bounding box is represented by a vector of $5 + C$ parameters where C is the number of classes. The first five parameters are the following: the coordinates x, y of the center of the box, the width w and height h of the box, and the probability p that the bounding box contains an object. The last C parameters are the class probabilities $P(Class_i|Object)$ representing the probability that the grid cell contains an object of class i given the fact that it contains an object. The predictions are therefore encoded as three dimensional tensor $N \times N \times (5 + C)$. To build a training dataset given a set of images, a bounding box is associated to each significant object and associated to the grid cell containing the center of the bounding box. An issue that may arise from this encoding is that, since an object is likely to span over several grid cells, the same object may be detected by the model for several grid cells, then leading to several bounding boxes around the same object. To solve this issue,

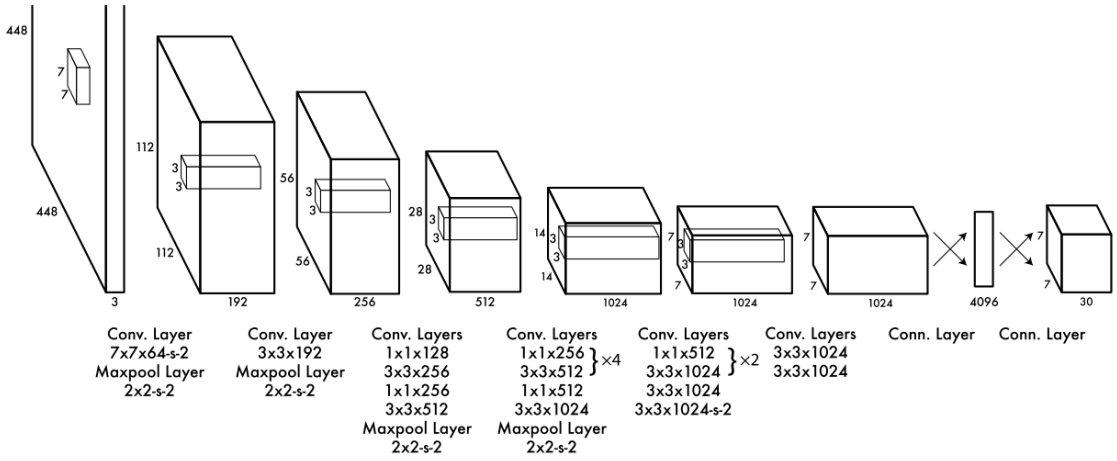


Figure 2.7: Architecture from [39]. The detection network has 24 convolutional layers followed by 2 fully connected layers.

it is possible to use a method called non-maximum suppression presented in Algorithm 1.

Algorithm 1 (Non-maximum suppression)**.**
Inputs
The bounding boxes predicted by the model associated with their respective class
A threshold thr
Outputs
The set R of remaining bounding boxes
Algorithm

```
R = {}
for every class c among the C classes do
    Let S be the set of all bounding boxes associated to class c
    while S is not empty do
        Let b be the box from S with the highest probability p
        S = S \ {b}
        R = R ∪ {b}
        for every box b' in S do
            if the intersection over union of b and b' is greater than thr then
                S = S \ {b'}
            end if
        end for
    end while
end for
```

The idea of non-maximum suppression is simply to keep only the box with the highest probability when several boxes are overlapping. The notion of overlapping is defined through intersection over union. The intersection over union of two boxes is the ratio between the area of the intersection of the boxes over the area of their union.

Since this is not an objective of this report, we will simply sketch the principles of the

Convolutional Neural Networks (CNN) models used to implement the YOLO algorithm. An input image is described as $M \times M \times K$ tensor where M is the width and the height of the image (we consider square image for simplicity) and K is the number of channels of the image (generally equal to three, for the three RGB channels). As shown in Figure 2.7, convolutional and pooling layers then transform this tensor into successive tensors with different dimensions, finally outputting a $N \times N \times (5 + C)$ tensor which can be interpreted as a grid containing bounding boxes as explained above. The loss between the output tensors and the true tensors is then computed and backpropagated to update the parameters of the network. In [39], the authors introduce a special loss for the YOLO algorithm, which is a modified Mean Squared Error. We provide a simplified version of this loss in Equation 2.12. $\mathbb{1}_i^{obj}$ is equal to 1 when the grid cell i contains an object in the true tensor and 0 otherwise. We have that $\mathbb{1}_i^{noobj} = 1 - \mathbb{1}_i^{obj}$. The constants λ_{coord} and λ_{noobj} aim at weighting the various terms of the loss. λ_{coord} is generally set above 1 to give a higher weight for localization compared to classification. λ_{noobj} is set below 1 to reduce the impact of the probability of object in grid cell where there is no object. Indeed, the majority of grid cell do not contain object hence pushing the probability of object toward zero and creating instability with respect to grid cells actually containing objects. The errors on widths w and heights h of the bounding boxes are computed using square roots in order to penalize more the errors on small boxes with respect to large boxes. Finally, it should be noticed that the prediction of position, dimension and class probabilities are only taken into account in grid cells actually containing objects.

$$
\begin{aligned}
L(Y, \tilde{Y}) = \quad & \lambda_{coord} \sum_{i=0}^{N^2} \mathbb{1}_i^{obj} [(x_i - \tilde{x}_i)^2 + (y_i - \tilde{y}_i)^2] \\
+ \quad & \lambda_{coord} \sum_{i=0}^{N^2} \mathbb{1}_i^{obj} [(\sqrt{w_i} - \sqrt{\tilde{w}_i})^2 + (\sqrt{h_i} - \sqrt{\tilde{h}_i})^2] \\
+ \quad & \sum_{i=0}^{N^2} \mathbb{1}_i^{obj} (p_i - \tilde{p}_i)^2 \\
+ \quad & \lambda_{noobj} \sum_{i=0}^{N^2} \mathbb{1}_i^{noobj} (p_i - \tilde{p}_i)^2 \\
+ \quad & \sum_{i=0}^{N^2} \mathbb{1}_i^{obj} \sum_{c \in C} (P_i(c) - \tilde{P}_i(c))^2
\end{aligned}
\tag{2.12}
$$

Optimization in Deep Learning

This paragraph provides a general overview of the classical optimization techniques used to train deep learning models. More specifically, we will focus on first order gradient descent methods, from Stochastic Gradient Descent (SGD), its variants with momentum, then adaptive learning rates leading to the Adam algorithm. All the references of the methods mentioned in this paragraph can be found in chapter eight of [16].

Let's consider the supervised and unregularized machine learning problem. The data (x, y) are generated according to a distribution p_{data}. The final goal is to minimize the expected generalization error (also called the risk) provided in Equation 2.13, where f is the model using the data x and the parameters θ to predict y and L is a loss function. Since the machine learning model has only access to a finite training set, we only know the empirical distribution $\tilde{p}_{data}$ over the training set. Hence, the objective of the machine learning model will be to minimize the empirical risk given in Equation 2.14, where m is the size of the training set.

$$J^*(\theta) = \mathbb{E}_{(x,y)\sim p_{data}} L(f(x, \theta), y) \tag{2.13}$$

$$J(\theta) = \mathbb{E}_{(x,y)\sim \tilde{p}_{data}} L(f(x, \theta), y) = \frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}, \theta), y^{(i)}) \tag{2.14}$$

When the error function L features bad optimization properties, it can be replaced by a surrogate loss function. This is the case in a binary classification task where the original loss function takes discrete values 0 or 1, and is not differentiable. In this example, the loss function can be replaced by the negative log-likelihood which will estimate the probability of belonging to each class. Another common issue of empirical risk minimization is overfitting. If the model is able to memorize the training set, it will achieve a very low value on $J(\theta)$ but will not be able to generalize well on $J^*(\theta)$. To avoid this issue, a machine learning algorithm will halt using an early stopping criterion generally based on the performances over the real error function. This means that, contrary to classical optimization, a machine learning algorithm will not converge to a local minimum (the final gradients may be very large).

We will now briefly review some machine learning algorithms. First, the Stochastic Gradient Descent (SGD) uses an estimate of the gradient as a descent direction. The main difference with a classical gradient descent is that, at each iteration, the SGD only considers a *minibatch* of m training example. Indeed, if the examples of the minibatch are drawn i.i.d. from the distribution p_{data}, then the average gradient over the minibatch is an unbiased estimate of the true gradient of the underlying expected generalization error $J*$. Of course, if the algorithm is trained over several *epochs*, meaning that the algorithm will pass through the whole training set several times, then we will obtain estimates of the gradient of the empirical risk J and not $J*$, since the minibatches will then be drawn from $\tilde{p}_{data}$. The main advantages of SGD is that the computational time of each iteration does not depend on the size of the full training set. When the size of the minibatch is the size of the training set, then the algorithm is called batch gradient descent. When the minibatch contains only one training example, it is referred as online gradient descent. SGD is presented in Algorithm 2.

Algorithm 2 (Stochastic Gradient Descent)**.**
Inputs
Learning rate schedule $\epsilon_1, \epsilon_2, \cdots$

Initial parameter θ
Algorithm
$k = 1$
while stopping criterion not met **do**
Sample a minibatch from the training set $\{(x^{(1)}, y^{(1)}), \cdots, (x^{(m)}, y^{(m)})\}$
Compute gradient estimate $\tilde{g} = \frac{1}{m} \sum_i \nabla_\theta L(f(x^{(i)}, \theta), y^{(i)})$
Apply update $\theta = \theta - \epsilon_k \tilde{g}$
$k = k + 1$
end while

Since the random sampling of the minibatch introduces noise in the estimaton of the gradient, the SGD algorithm has to reduce the learning rate ϵ at each iteration to converge. Sufficient conditions for the convergence of SGD are the following:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \tag{2.15}$$

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty \tag{2.16}$$

A first approach to accelerate learning with SGD is the method of momentum. The momentum algorithm consists in introducing a momentum p corresponding to an exponentially decaying moving average of past gradients. This momentum v is the velocity at which the parameters moves through the search space. This is an analogy with the physical laws of motion where the negative gradients are seen as forces acting on a particle moving in the search space. The convergence is accelerated by reducing the variance in the SGD and by reducing the effect of poor Hessian conditioning. We recall that poor Hessian conditioning corresponds to a high ratio between the highest and the lowest eigenvalues of the Hessian matrix, and leads to oscillating behavior when using SGD. Considering Algorithm 2, the momentum algorithm consists in replacing the update phase by the two steps presented in Equations 2.17 and 2.18.

$$v = \alpha v - \epsilon \tilde{g} \tag{2.17}$$

$$\theta = \theta + v \tag{2.18}$$

where $\alpha \in [0, 1)$ is an hyperparameter corresponding to the strength of the exponential decay. The larger α is, the more past gradients will impact the current step. This hyperparameter can be seen as a viscous drag.

Another approach to accelerate learning are the adaptive learning rates algorithms. The general idea is to automatically adapt a learning rate for each direction of the search space during training. AdaGrad scales the learning rates proportionally to the inverse square root of the sum of the past squared gradients, such that greater progress is achieved in the direction of less steep slope. The squared gradients are accumulated

in a parameter r initialized to zero. The update phase of SGD becomes Equations 2.19 and 2.20.

$$r = r + \tilde{g} \times \tilde{g} \tag{2.19}$$

$$\theta = \theta - \frac{\epsilon}{\delta + \sqrt{r}} \times \tilde{g} \tag{2.20}$$

where $\times$ is the element-wise multiplication, ϵ is a global learning rate, and δ is a small constant for numerical stability. The main issue of AdaGrad is that it accumulates the squared gradients from the start of the algorithm which induces a fast decrease of the learning rates, eventually slowing the learning during the last iterations. In fact, AdaGrad achieves fast convergence in the convex setting but is not well suited for nonconvex optimization. To address this issue, the RMSProp algorithm modifies the AdaGrad algorithm by changing the accumulation of squared gradients by a exponentially weighted moving average, enabling the algorithm to forget the first gradients that may be associated to very different structures. In RMSProp, the update phase of the SGD is replaced by Equations 2.21 and 2.22.

$$r = \rho r + (1 - \rho)\tilde{g} \times \tilde{g} \tag{2.21}$$

$$\theta = \theta - \frac{\epsilon}{\sqrt{\delta + r}} \times \tilde{g} \tag{2.22}$$

where ρ is an hyperparameter controlling the length scale of the moving average. RMSProp can also be hybridized with momentum method. This leads to a new adaptive learning rate optimization method called Adam (for "adaptive moments"). The Adam algorithm computes estimates of the first-order moment s and uncentered second-order moment r of the gradients, with exponential weighting. The momentum is not applied to the rescaled gradients (as with RMSProp + momentum) but directly incorporated during the first-order moment estimation. Adam also includes a bias correction of both first-order and second-order moments. The update phase of Adam is provided in Equations 2.23 through 2.23.

$$s = \rho_1 s + (1 - \rho_1)\tilde{g} \tag{2.23}$$

$$r = \rho_2 r + (1 - \rho_2)\tilde{g} \times \tilde{g} \tag{2.24}$$

$$\hat{s} = \frac{s}{1 - \rho_1^k} \tag{2.25}$$

$$\hat{r} = \frac{r}{1 - \rho_2^k} \tag{2.26}$$

$$\theta = \theta - \epsilon \frac{\hat{s}}{\sqrt{\delta + \hat{r}}} \tag{2.27}$$

where ρ_1 and ρ_2 are the exponential decay rates for moments estimates. Note that in Equation 2.27, the operations of the updating term are applied element-wise. The Adam algorithm is currently one of the most widely used optimization algorithm for deep learning applications. In particular, all the experiments conducted in this report will used the Adam algorithm to train the various models.

2.4 Trajectory Prediction with Machine Learning approaches

Trajectory prediction is a classical problem in ATM research since most of operational applications require at some point a precise and reliable trajectory prediction. Trajectory prediction approaches can be divided into two main categories: deterministic and probabilistic. Deterministic approaches relies on a specific aerodynamic model to estimate the state of the aircraft and then propagate the states into the future, for example by using Kalman filter ([3]). The main drawback of the deterministic approach is that it struggles to take into account uncertainties coming from future winds or pilot actions and hence is prone to degraded precision accuracy if used over a long period of time. The deterministic approach is then more suited for the prediction of specific flight phases.

On the other hand, the probabilistic approach relies on statistical models able to learn the trajectory of aircraft from historical datasets. A first category of probabilistic approaches requires the dataset to cover all possible trajectories. These methods select one predicted trajectory among all possible using relevant features. For example, the authors of [1] trained a Hidden Markov Model (HMM) on a historical trajectory and weather dataset. Another probabilistic approach consists in providing the probability distribution of the aircraft position. For example, the authors of [29] uses the flight plan and the current position of the aircraft to build a discrete time probabilistic model to predict the future position of the aircraft. This model is then applied to estimate the probability of conflicts.

Finally, it is also possible the trajectory prediction task as a pure regression task. In [8], the authors trained a Generalized Linear Model (GLM) to use wind and aircraft initial state to predict the Estimated Time of Arrival (ETA) over several waypoints in the terminal area. In [27], the full 4D trajectory is predicted using an encoder-decoder architecture of RNNs, as presented in section 2.3. The sequence input of the encoder network is the flight plan defined as a sequence of 2D positions. The sequence input of the decoder network is the concatenation of the actual 3D position and speed along with weather features. These weather features are extracted with convolutional layers (CNN) from a local box surrounding the actual position and containing wind, temperature and convective weather information. The trajectory points are sampled at a constant time rate, hence integrating the time component of the 4D trajectory. The decoder network does not output directly a prediction of the 3D position for the next time step, but rather a prediction of the parameters of a Gaussian Mixture Models (GMM) representing the distributions of the position and speed of the aircraft. During the inference process,

this GMM model is used along with a beam search procedure to produce the most likely trajectory from the last known position until the arrival airport. The predicted trajectory is also smoothed using Adaptive Kalman Filter. The RNN model is trained on historical trajectories between an airport pair along with the corresponding weather data. Since the decoder network predicts the mean and covariance matrix of a GMM, the model is able to provide an evaluation of the uncertainty at each point of the predicted trajectory. In [34], the authors also apply RNN architecture to predict 4D trajectories, but with a different approach than the one proposed in [27]. Instead, they only use one network of LSTM where the input sequence is the flight plan positions regularly resampled in time over the whole trajectory. The input sequence also includes weather features extracted with a convolutional network similar to the one used in [27]. Then, at each time step, the LSTM network has to predict the true 3D position using the planned position and the weather information.

We will take inspiration of the work of [27] and [34] to provide a trajectory prediction based on RNN networks in chapter six, which will be the basis to compute a complexity metric using the future predicted trajectories in a given portion of the airspace.

2.5 Data sources

Three sources of data are used throughout this project.

In chapters three, four, and five, we use a dataset of simulated trajectories representing a characteristic day of traffic over the French airspace. A trajectory is a sequence of aircraft states. Each aircraft state consists in the following elements: a timestamp, an aircraft ID, a latitude, a longitude, an altitude, a true air speed, a heading, and a rate of climb. As it is detailed in chapter three, a complexity value is then added to each of these aircraft states. This complexity value can be computed using various methods, including the linear dynamical system approach presented in section 2.1. The trajectories are created by inputting flight plans to a simulator, then computed with optimal vertical profiles (no flight levels) and without any deconflicting actions. The atmosphere is assumed to be in ISA conditions with no wind. The final dataset is composed of 8011 simulated trajectories spread along 3025 time steps (1 time step equals 15 seconds).

In chapter six, we use a different software to generate trajectories between pair of airports taking into account wind information. This simulator, called the Aircraft Arithmetic Simulator, implements controlled equations of motion using a physical model based on total energy variation similar to the one underlying BADA. The weather module is able to take into account temperature, pressure, and wind at any point of the simulation state, but we will only add the wind information, while pressure and temperature will be set to their ISA values. The simulator takes in input a flight plan represented by a sequence of 2D points along with a cruise flight level, a cruise speed, an aircraft type, and an initial mass. The output is a trajectory defined as a sequence of aircraft state, with each state composed of a callsign, a timestamp, a latitude, a longitude, an altitude, a heading, a rate of climb, a true air speed, and a mass.

To generate the trajectories used in chapter six, we use the Aircraft Arithmetic Simulator with three different flight plans between TLS and CDG, along with various wind conditions. The wind data are taken from the Météo-France public data[1]. We retrieve 8 grib2 files covering two days of weather information. After extraction of the data, we obtain a 33×23 grid with resolution 0.5°, with longitudes from -6° to 10° and latitudes from 41° to 52°. The vertical plane is decomposed into 28 isobaric levels ranging from 100,000 Pa (corresponding to sea level) to 1,000 Pa. Figure 2.8 presents the wind information at the level 90,000 Pa, at a given date.

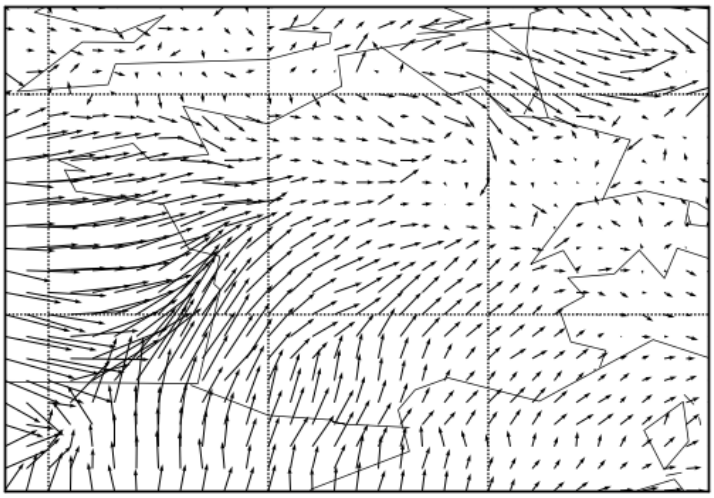


Figure 2.8: Example of wind data on 08/27/2020 12:00 at level 90,000 Pa.

[1] `https://donneespubliques.meteofrance.fr/`

Chapter Three

Sequence to vector model

3.1 Methodology

The objective of the report is to build a machine learning model able to predict congested areas in a time horizon of 40 minutes. In this chapter, this objective will be defined as a supervised learning regression task. To achieve this, a training set has to be built using pairs of training inputs/outputs. Let us first describe the training outputs. Since we want to predict the congested areas, the training outputs have to provide a measure of the congestion in each point of the airspace at a given time. This measure of congestion is achieved with a complexity metric, as the ones described in subsection 2.1. In this chapter, the complexity metric presented in [14] and referred as "linear dynamical system" is used.

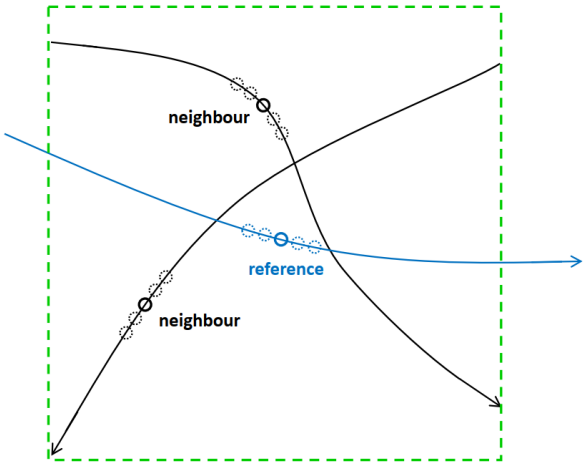


Figure 3.1: Neighborhood of the reference aircraft. The dotted circles represents the extension of the aircraft positions. Source: [14].

This metric is an intrinsic metric providing a measure of the traffic complexity without reference to the perceived workload, and only linked with the geometrical organisation of the trajectories. As described in [14], this metric can be adapted to compute a measure

of the complexity in the neighborhood[1] of an aircraft, at a given time. A filter is applied to consider only the flights that may interact with the reference aircraft. For example, another aircraft in level flight vertically separated with the reference aircraft by 1000 ft (in RVSM) will not interact with the reference aircraft, and should not be considered in the metric computation. After this filtering, the selected aircraft positions are extended by adding ten forward positions (separated by one time step) and ten backward positions (see Figure 3.1). The aim of this extension is to take into account the uncertainties on the true aircraft's positions. Each of this added points is treated as a "new aircraft". Finally, a linear dynamical system $\dot{X} = AX + b$ is fitted to the positions and speeds of these aircraft. The complexity metric C is then defined as the sum, in absolute value, of the negative real part of the eigenvalues of the matrix A as shown in the following expression $C = -\sum_{\mathrm{Re}(\lambda(A))<0} \mathrm{Re}(\lambda(A))$.

To build the training outputs, we defined for each time step t a $N \times N$ matrix $y[t]$. This matrix is superimposed over the airspace, such that each element of the matrix covers a small area of the airspace. To avoid a too high output dimension, we do not consider the altitude axis. Hence, each area covers all flight levels and is only defined by the latitude and longitude of its center point. Then, for a given area, the corresponding element of the matrix takes the maximum value (or the mean value) of the complexity metric of the aircraft inside this area at time step t. The complexity values are then scaled by a logarithmic function $x \to \log(1 + x/thr)$ to obtain a more uniform distribution (the parameter thr is a threshold). The heat map in Figure 3.2 provides an example of a matrix $y[t]$. The $y[t]$ matrices are then flattened as N^2 vectors to be used in the machine learning model.

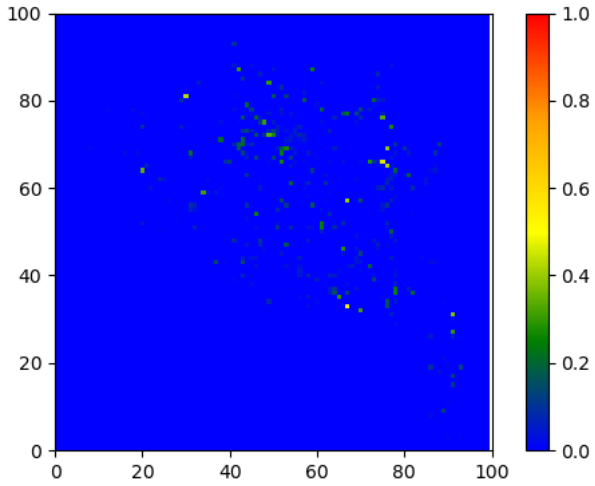


Figure 3.2: Example of visualisation with $N = 100$, $thr = 10^{-2}$, and linear normalization between 0 and 1. The color scale indicates the maximum complexity metric value in each cell.

Let us know focus on the training inputs. We will defined three different types of

[1]The neighborhood is defined as a $24,8NM \times 24,8NM$ box horizontally and $\pm 30FL$ vertically, centered on the reference aircraft.

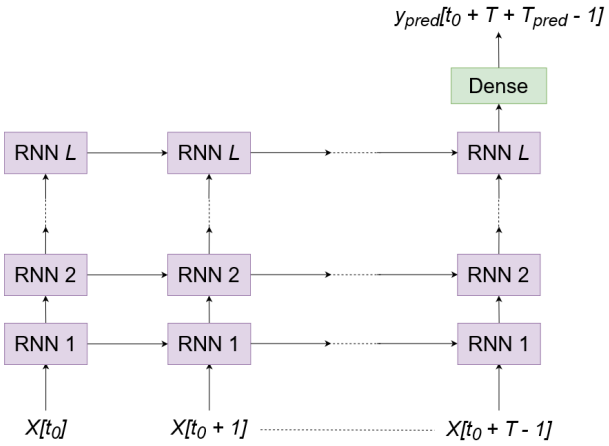


Figure 3.3: Basic RNN model unrolled through time

input features and compare their performances to predict the future congested areas. The first type of inputs (referred as Type I) is simply a sequence of flattened $y[t]$ matrices for t ranging from t_0 and $t_0 + T - 1$. Hence, the input vectors are $x[t_0] = y[t_0]$ until $x[t_0 + T - 1] = y[t_0 + T - 1]$ while the corresponding training output is $y[t_0 + T + T_{pred} - 1]$ (where T_{pred} is the number of time steps between the end of the input sequence and the prediction). The second type of inputs (Type II) will use the aircraft states. For a given point in an aircraft trajectory, we define the aircraft state as $state = \begin{bmatrix} longitude & latitude & altitude & ground\_speed & heading & climb\_rate \end{bmatrix}$. At a given time t, the input vector $x[t]$ is built by concatenating all the aircraft states currently in the airspace, ordered by increasing longitude (and by increasing latitude if the longitude is the same), such that the positional information is preserved in the structure of the inputs. To get a fixed-sized input vector, the dimension of $x[t]$ is set to the maximum number of simultaneous aircraft states in the dataset (multiplied by the length of an aircraft state). If the number of states is strictly lower, the remaining elements of $x[t]$ are padded with zeros. Finally, Type III input is similar to Type II but with a different aircraft state, referred as an extended state, and defined as $extended\_state = \begin{bmatrix} longitude & latitude & altitude & ground\_speed & heading & climb\_rate & C & a_{1,1} & \cdots & a_{3,3} \end{bmatrix}$ where C is the complexity metric defined above and the $a_{i,j}$ are the elements of the A matrix of the linear dynamical system previously mentioned. We can know describe the training set as pairs of (input sequence, output vector) $= \left(x[t_0] \quad \cdots \quad x[t_0 + T - 1] \quad , \quad y[t_0 + T + T_{pred} - 1] \right)$ where x can either be Type I, II or III input features. In the following, the dataset will be randomly divided between training examples (90% of the dataset) and validation examples (10% of the dataset).

We now describe the models that are used in this chapter. Since the input features are defined as sequences, it is natural to introduce RNN models. We start with a basic RNN model presented in the Figure 3.3. This is a many-to-one network, as introduced in section 2.2. The input features are fed into L successive recurrent layers (which can either be LSTM or GRU). After the last time step, the hidden state of the L-th recurrent layer is fed into a dense layer to output a prediction $y_{pred}[t_0 + T + T_{pred} - 1]$. The selected error function is the Mean Squared Error between the prediction and the ground-truth

$y[t_0 + T + T_{pred} - 1]$. From this basic architecture, it is possible to modify and tune the models as well as the processing of the data with various hyperparameters. Hyperpamareters are parameters that have to be chosen before the optimization process, contrary to the weights and biases of the network that are fixed during the optimization process. Concerning the model itself, we consider the following hyperparameters: the class of recurrent layer (LSTM or GRU), the number of layers (depth of the network) and the dimension of the hidden state of each layer (width of the layer). We will also investigate the use of transposed convolution layers after the dense layer. Since the target output is the matrix $y[t]$ containing the complexity values, it may be relevant to reconstruct an image by taking into account the spatial structure of such images. This is possible with transposed convolution layers, which consist in using the convolution operation (as with classical convolutional layers) to output larger images from smaller input images. Concerning the optimization algorithm (see section 2.3), we consider the number of epochs, the learning rate, the minibatch size (later simply referred as 'batch size'), along with two regularization techniques: L2 regularization and dropout. L2 regularization consists in adding a regularizing term $\frac{\lambda}{2}\|w\|^2$ to the loss function, where w denotes the vector of the weights of the network. This forces the optimization algorithm to choose small values for the weights and hence to choose a parsimonious model and avoid overfitting. In this case, the hyperparameter is the constant λ (the more we increase the value of λ, the more the regularization term will impact the loss function with respect to the error term). Dropout consists, at each iteration of the optimization, to randomly suppress a given proportion of neuron (meaning to force their output to be zero). Once again, this method aims at forcing the algorithm to choose a more flexible and parsimonious model and avoid overfitting. In this case, the hyperparameter is the proportion of neurons that will be randomly shut down at each iteration. We also investigate the gradient norm scaling technique. This technique consists in imposing an upper bound on the norm of the gradients computed during backpropagation, thus preventing the exploding gradient problem (values of the gradient diverging, in particular when backpropagating through a recurrent network with lots of time steps). Here, the hyperparameter is the upper bound of the norm of the gradient. Concerning the processing of the data, we consider the length of the input sequence, as well as a Gaussian smoothing of the complexity matrices $y[t]$. This "Gaussian smoothing" consists in applying a 3×3 approximative Gaussian kernel $\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$ on the matrix to smooth the values of the complexity metric across the airspace. These smoothed data are then used to train the model. It can be expected than it is easier for a model to extract relevant patterns from smoother data.

3.2 Results

In this section, we provide the results for the basic model introduced in section 3.1 along with experiments on the various hyperparameters. All the models of this report are implemented with the Keras library using the TensorFlow 2 backend.

The following notations are used for the hyperparameters. Concerning the architecture of the model: we consider the length of the input sequence T, the class of recurrent layers rl (LSTM or GRU), the number of layers L and dimension of the hidden states of each layer. Concerning the optimization algorithm, we use the Adam algorithm with parameters $\rho_1 = 0.9$, $\rho_2 = 0.999$ and learning rate decay $d = 10^{-2}$ (these values are used for all the experiments of this report). The learning rate ϵ and batch size bs are evaluated. The impact of the L2 regularization coefficient λ and the dropout rate dr is also considered. Finally, we evaluate the gradient norm scaling value cn, which is a threshold for the gradient norm aiming at avoiding the exploding gradient problem. To evaluate the impacts of these hyperparameters, we chose a default set of hyperparameters, and then change one hyperparameter's value at a time from the default values. The metric used to measure the model's performance is the average of the validation loss over the ten last epochs. The default hyperparameters values are the following: $T = 160$, $rl = GRU$, $l = [512, 512, 512, 512]$ (meaning 4 recurrent layers with 512 hidden units in each one of them), $bs = 64$, $\epsilon = 10^{-4}$, $dr = 0.1$, $\lambda = 10^{-5}$, $cn = 1$. Since the target outputs are nonnegative, we can use the ReLU activation function for the Dense layer (the ReLU function is the positive part function $x \to \max\{x, 0\}$). The model is trained over 200 epochs with a prediction time $T_{pred} = 160$ (corresponding to 40 minutes). The training of one model (200 epochs) requires about two hours on a server with eight GPU RTX 2080 Ti.

In the Table 3.1, we present the results of the hyperparameters search. In the left column, only the modified hyperparameter is shown with its new value. The hyperparameters having a significant impact on the model's performance are: the learning rate ϵ which slows down training when reduced, the L2 regularization which also harms the performances, the batch size bs which slows down training when increased (because the parameters updates are less frequent), and the choice of recurrent layer (LSTM seems to have slightly better performance). For the next experiments, the following values will be used: $\epsilon = 10^{-3}$, $bs = 32$, $rl = LSTM$. No regularization techniques will be implemented (no L2 regularization and no dropout). The other hyperparameters are kept to their default values.

Three models are now trained on 500 epochs with each input types (I, II and III). Loss curves for each model are shown in Figure 3.4. In this Figure, the values of the loss over the training set (in blue) and over the validation set (in orange) are plotted with regards to the number of epochs. Each model seems to achieve similar performances, with a final validation loss in the order of $7 \cdot 10^{-3}$. Type I input (using directly $y[t]$ matrix as inputs) seems to achieve slightly better performance, which was expected since in this case the inputs and outputs share the same structure. However, Type I input requires the computation of the complexity metric before making predictions, while Type II input needs only the aircraft states. Type III input seems to bring no improvement over Type II input.

In the following, we provide the results of the experiments with Gaussian smoothing. Here, we use the Type II input along with a different model: 3 LSTM layers with hidden dimension of 512, and 2 Dense layers with respective output dimensions 1024 and 10000. The loss function used in the optimization algorithm is a weighted MSE defined as:

Hyperparameters values	Validation loss ($\times 10^4$)
Default	180
$T = 80$	182
$T = 320$	188
$rl = LSTM$	154
$l = [256, 256, 256, 256]$	168
$l = [1024, 1024, 1024]$	191
$l = [512, 512, 512, 512, 512, 512]$	213
$l = [1024, 256, 256, 1024]$	189
$bs = 32$	158
$bs = 128$	232
$\epsilon = 10^{-5}$	720
$\epsilon = 10^{-3}$	112
$dr = 0$	176
$dr = 0.3$	202
$\lambda = 0$	102
$\lambda = 10^{-4}$	312
$cn = 10^{-1}$	189
$cn = 10$	198

Table 3.1: Validation loss for various hyperparameters values

$L(y, \hat{y}) = \frac{\sum_i e^{k \cdot y_i}(y_i - \hat{y}_i)^2}{\sum_j e^{k \cdot y_j}}$ with $k = 5$. The rational behind this loss function is to increase the error value when the true complexity value is high, and hence to force the model to provide its most accurate predictions for high complexity values. The performances of the model are measured by the weighted MSE over the validation set. When using the Gaussian smoothing on the output matrices $y[t]$ we obtain a validation loss of 3×10^{-5}, while this validation loss reaches 69×10^{-5} when using the original dataset (without the Gaussian smoothing). This result confirms that the learning task is easier when using smoothed target outputs. Figure 3.5 provides an example of the validation set showing the prediction of the model and the true output with Gaussian smoothing.

To test the transposed convolution, we use a model with the Type II input, 3 LSTM layers with hidden dimension 512, and 1 Dense layer followed by 2 transposed convolution layers. The first transposed convolution layer transforms a $24 \times 24 \times 1$ input tensor into a $49 \times 49 \times 64$ tensor. The second one takes the $49 \times 49 \times 64$ as input and outputs the final prediction as a $100 \times 100 \times 1$ tensor. Here, the validation loss is 89×10^{-5} to be compared with the value 69×10^{-5} obtained without the transposed convolution layers. This seems to show that the transposed convolution layers harms the performances of the model.

In this chapter, we have suggested a direct approach to complexity prediction. The complexity values in 40 minutes are predicted with a basic RNN model taking as input the states of the aircraft at each timestamp from the last minutes. The model seems to be able to predict the location of the congested areas. However, an evaluation by controllers is needed to quantify the efficiency and usefulness of this approach, in particular its capacity to help anticipate traffic bursts and increase the airspace capacity.

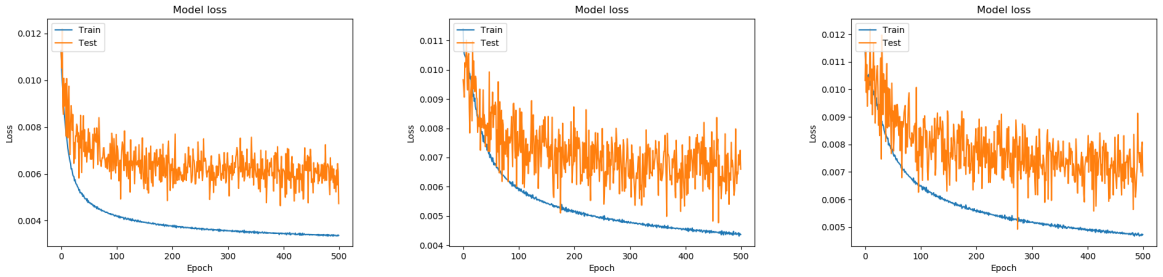


Figure 3.4: Training and validation (referred as test) losses per epoch using Type I input (left), Type II (center) and Type III (right)

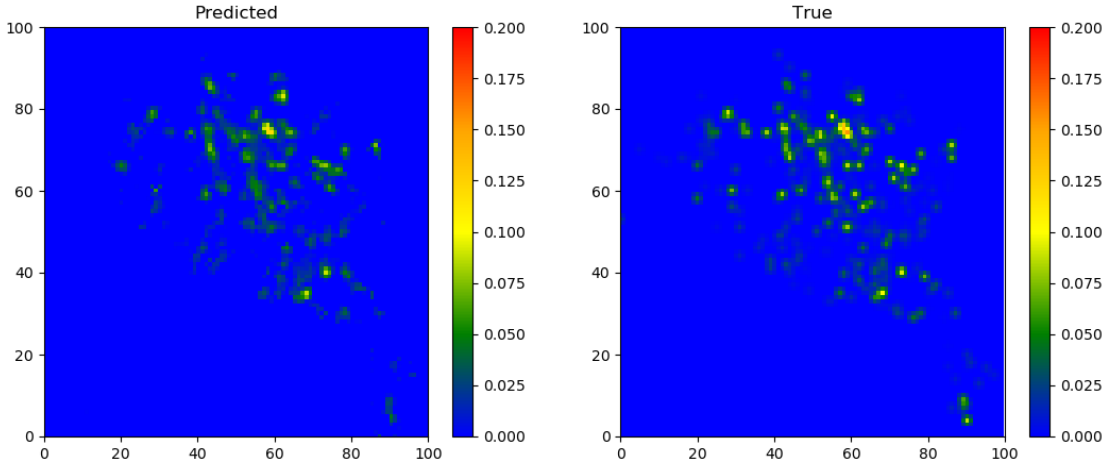


Figure 3.5: Validation example prediction (left) compared to the true output (right) with Gaussian smoothing

Chapter Four

Encoder-decoder model

4.1 Methodology

In this chapter, we keep the same approach as introduced in chapter three but with different and more sophisticated models. The models of this chapter fit in with the general framework of encoder-decoder networks presented in section 2.3. Before diving into the details of the model, we provide some insights of the dataset and the corresponding task that the model has to address.

We use a similar dataset as in the previous chapter composed of inputs/outputs pairs. In this chapter, we will only use sequences of aircraft states (referred as Type II inputs in the previous chapter). Once again, the outputs are defined as $N \times N$ matrices covering the airspace. Each element of these matrices is a measure of the complexity in a given area at a given time. This measure of the complexity can be computed using various complexity metrics. One of them is the "linear dynamical system" metric used in the previous chapter. In this chapter, we also use two other metrics, "robust convergence" and density. As with the linear dynamical system metric, the robust convergence metric assigns a scalar to each aircraft at each timestamp. For each aircraft, we consider the relative speed of the aircraft with each aircraft in a given neighborhood. These relative speeds are then aggregated to get a single number quantifying the level of convergence. The aircraft that are level off and separated vertically with the considered aircraft are not taken into account, while aircraft in evolution that may interact with the considered aircraft are taken into account. The density metric, as used in this report, is a very simple measure of complexity. Its consists in the total number of aircraft present in a given area at a given time.

The model implemented in this chapter is represented in Figure 4.1. Contrary to the model of the previous chapter, this model predicts a sequence of complexity matrices (and not a single matrix for the 40th minute after the last input sequence's element). Hence, this model is a sequence-to-sequence network. Let's first consider the encoder network (left part of the figure). The encoder network takes in input a sequence of vectors of aircraft states $X[t_0], \ldots, X[t_0 + T + 1]$ of length T. We recall that each $X[t]$ is

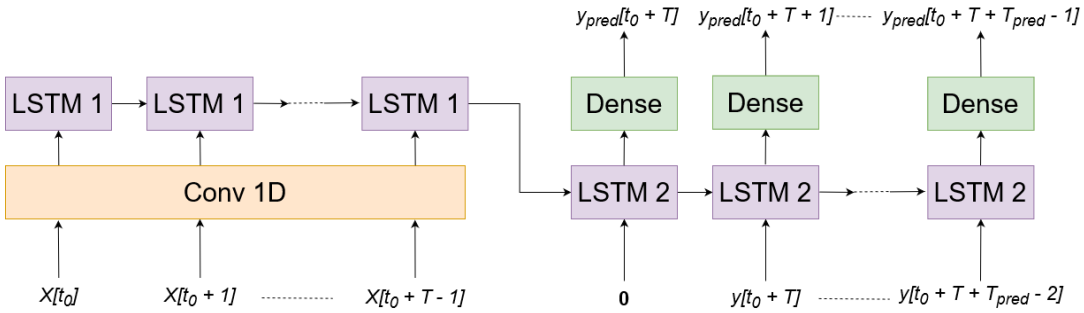


Figure 4.1: Encoder-decoder model with 1D convolutional layer

a vector where all the aircraft states present in the airspace at timestamp t are concatenated. $X[t]$ is then padded with zero such that all the elements of the input sequence have the same dimension. The input sequence is then fed into one or several 1D convolutional layers (depicted in Figure 4.2). A 1D convolutional layer aggregates a small subsequence of the input sequence (the length of this subsequence depends on the width of the kernels of the layer, which is an hyperparameter of the network) such that each element of the sequence obtained in output of the 1D convolutional layer, contains information from several successive timestamps, instead of looking at only one timestamp oblivious of the time dependence of the input sequence. More precisely, we have $y_{ij} = \phi(\sum_{k=-K}^{K} \sum_l w_{kl}^j x_{i+k,l})$ where y_{ij} is the j-th vector element of the i-th sequence element outputted by the 1D convolutional layer, and x_{kl} is the l-th vector element of the k-th sequence element of the input. w^j is the matrix of the j-th kernel, K is the width of the kernels, and ϕ is an activation function. If $i + k$ is lower than zero or greater than the input sequence's length, then $x_{i+k,l} = 0$ for all l. The output of the 1D convolutional layers is fed into several LSTM layers. The hidden state of the last LSTM layer obtained after feeding into the encoder network the last element of the input sequence is the encoding vector, which is the only information that will be passed to the decoder network (right part of the Figure).

The decoder network is composed of several LSTM layers (the hidden states of these layers are initialized with the encoding vector computed by the encoder network) followed by several Dense layers which finally output a vector of dimension N^2 corresponding to the matrix of complexity values. This is a simplification from the encoder-decoder framework described in section 2.3. In section 2.3, the output of the decoder network is described as a distribution over all possible elements of an output sequence. The final predicted sequence is then built by choosing the output sequence with the highest overall probability (given the input sequence), or at least using an heuristic like beam search allowing to get a good enough prediction in a reasonable computation time. This framework is more suited for classification tasks where the distribution is discrete (and finite). Even if it is possible to choose a family of continuous distributions to represent the set of possible output distributions of the decoder network for regression task, a simpler approach has been retained here. The network directly outputs a prediction $y[t]$ for the t-th element of the output sequence. If we denote the size of the output sequence by T_{pred}, this greedy approach can be seen as approximating

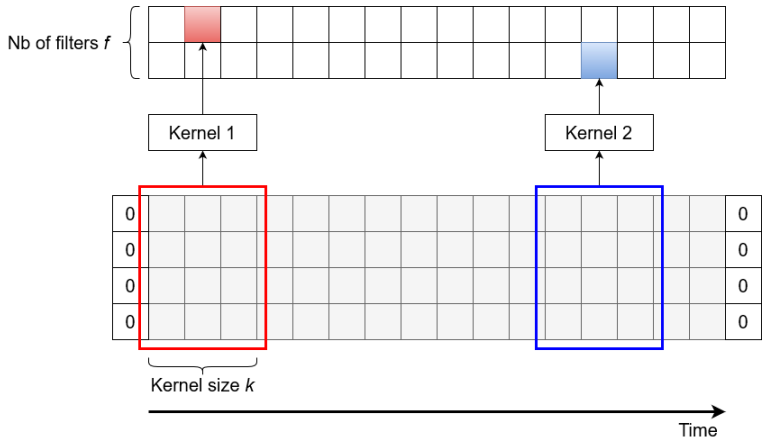


Figure 4.2: Illustration of a 1D convolutional layer. In this example, the input is a sequence of vectors of $\mathbb{R}^4$. The output is a sequence of vectors of $\mathbb{R}^2$, which means that the layer is composed of 2 filters, each one being composed of a kernel of width $k = 3$.

$\max_y P(y|x) = \max_y \prod_t P(y_t|y_1, \ldots, y_{t-1}, x)$ by $\prod_t \max_{y_t} P(y_t|y_1, \ldots, y_{t-1}, x)$. It is the same thing than using the beam search algorithm with a beam width of 1. A last aspect of the architecture of the model is that the decoder network is implementing a technique called "teacher forcing". This means that, during training, the decoder network takes as input the true output sequence, but shifted by one timestamp. In other words, at a given timestamp, the decoder network receives as input the output it should have predicted for the previous timestamp. During the inference process, the true output sequence is obviously not known. In this case, the decoder network simply takes as input the prediction it has made at the previous timestamp. In both cases, the first input to the decoder network is the null vector.

Several variations on the dataset or on the network are tested in this chapter. First, we consider the dataset where the complexity matrices $y[t]$ are smoothed using a Gaussian kernel as introduced in the previous chapter. We then investigate the effect of the extension of aircraft position in the computation of the linear dynamical system metric (this extension is presented in section 3.1). As a reminder, this extension consists in adding m points corresponding to future positions of each aircraft and m points corresponding to past position of each aircraft (the points are separated by one time step). These points are then treated as additional aircraft and aim at modelling the uncertainty of the aircraft's positions along the curvilinear abscissa. In particular, we will try to see if the choice of the parameter m affects the performances of the model.

Finally, we test the model on a different task. Rather than learning to predict the values of the complexity as a regression task, we consider the binary classification task consisting in predicting if the complexity in each area is above a given threshold or not. To train a model on this new task, we need to build a new dataset. To do so, we start with the dataset of continuous complexity values. We then choose a relative threshold η (for example $\eta = 5\%$) and set to 1 all the areas with complexity values greater than $(100 - \eta)\%$ of all the complexity values in the dataset, while the others are set to 0. The same encoder-decoder model is training on this new dataset, but with a different loss

function. Since each area corresponds to a binary variable, the classification task can be seen as a multi-label classification, where each area is a separated class which can be either present (the target is 1) or absent (the target is 0) in each complexity matrices $y[t]$. For multi-label classification, the suitable loss function is the binary cross-entropy $L(y, \hat{y}) = \sum_{c=1}^{N^2} y_c \log(\hat{y}_c) + (1 - y_c) \log(1 - \hat{y}_c)$ where y_c is the target output corresponding to the c-th area. During the tests with the binary classification task, we will test the performances using various complexity metrics (either linear dynamical system or robust convergence) and also investigating the structure of the inputs. Remember that in section 3.1, the type II inputs are defined as the concatenation of the states of all the aircraft present in the airspace at a given time step. These aircraft states are ordered according to their geographical position, more precisely by increasing longitude, and by increasing latitude if the longitude is the same). This ordering of the aircraft states into one input vector $x[t]$ will be referred as "by position" ordering. Another ordering, referred as "by aircraft" ordering, is also tested. In this case, each aircraft receives a fixed index in the $x[t]$ input vector when it enters the airspace. Hence, each aircraft state remains at the same location inside $x[t]$ for all ts for which the aircraft fly in the airspace. When an aircraft leaves the airspace, its corresponding state location is set to the null vector. When another aircraft enters the airspace, its receives the smaller available index. This structure "by aircraft" is supposed to help the network distinguish each individual aircraft, while the "by position" structure is assumed to provide the network with some information about the relative positions of the aircraft.

4.2 Results

We first provide the results for the basic encoder-decoder model. The hyperparameters used for the experiments are provided below. We begin by a description of the encoder network. Thee first layer is a 1D convolutional layer with a kernel of width $K = 3$ and $f = 512$ filters (the number of filters is the dimension of the output of the layer). This layer is followed by one LSTM layer with an hidden dimension of 128. The number of time steps for the encoder network is fixed at $T = 160$. Since 1 time step is 5 seconds, this corresponds to 40 minutes, such that the model has access to the last 40 minutes of traffic to compute its prediction. The decoder network is made of one layer of LSTM with hidden dimension 128 followed by one Dense layer with output dimension of 10000 (since our output is a $N \times N$ matrix with $N = 100$. All the variables of the dataset (from the input as well as from the output) are linearly normalize between 0 and 1. We use the ReLU activation function for the last Dense layer. The sequence predicted by the decoder network are set to $T_{pred} = 160$, which corresponds to predict the next 40 minutes of complexity values. Concerning the learning algorithm, we use the Adam algorithm with a learning rate $\epsilon = 10^{-3}$, $\rho_1 = 0.9$, and $\rho_2 = 0.999$. The loss function is the Mean Squared Error. The mini-batch size is set to $bs = 128$ and the model is trained over 100 epochs. The 8011 simulated trajectories of our dataset are spread across 3025 time steps. Since our model requires pair of sequences of respective length T and T_{pred} that follow one another, we can build $3025 - T - T_{pred} = 2705$ sequences. Among these 2705

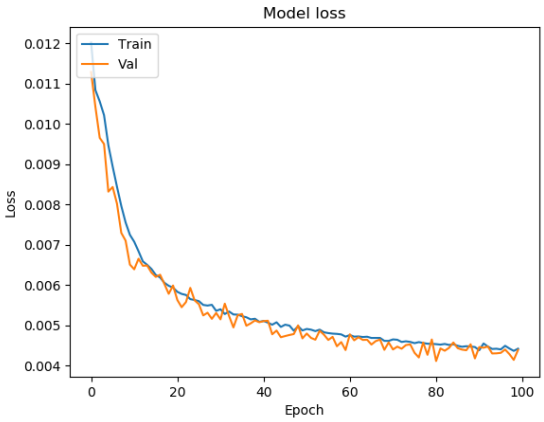


Figure 4.3: Training and validation losses with the basic encoder-decoder model.

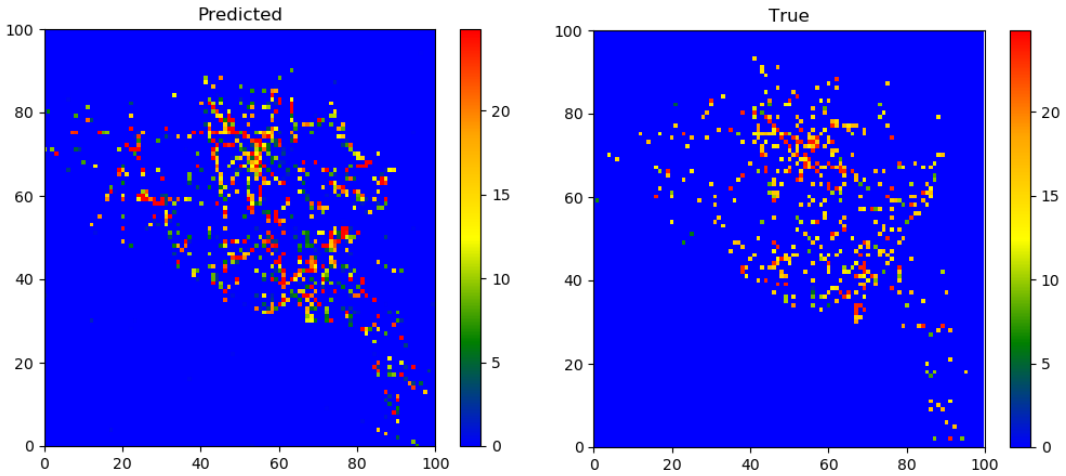


Figure 4.4: Training example. True value after 40 minutes (left) and predicted value (right)

sequences, 2434 are put in the training set (used to train the model), and 271 are put in the validation set (used to evaluate the generalization performances of the model).

Figure 4.3 illustrates the training and validation losses with the model described above. The loss curves provide the mean squared error for each epoch of the training, evaluated over the training set and the validation set. Both losses converge towards a MSE of 4.5×10^{-3}. This indicates that the model was able to learn from the data. Let's examine the performances of the model when using the inference process. Remind that the difference between the training and the inference process is that, during inference, the decoder network uses its own previous predictions as input rather than the true target (which is assumed to be unknown). In Figure 4.4, we show an example of prediction from the training set. We can visually check that the model is able to predict the true output in 40 minutes with a reasonable accuracy. Figure 4.5 shows the same type of prediction but using a validation example (an example that was not used to train the model). It can be seen that the model is less accurate with validation examples. It may

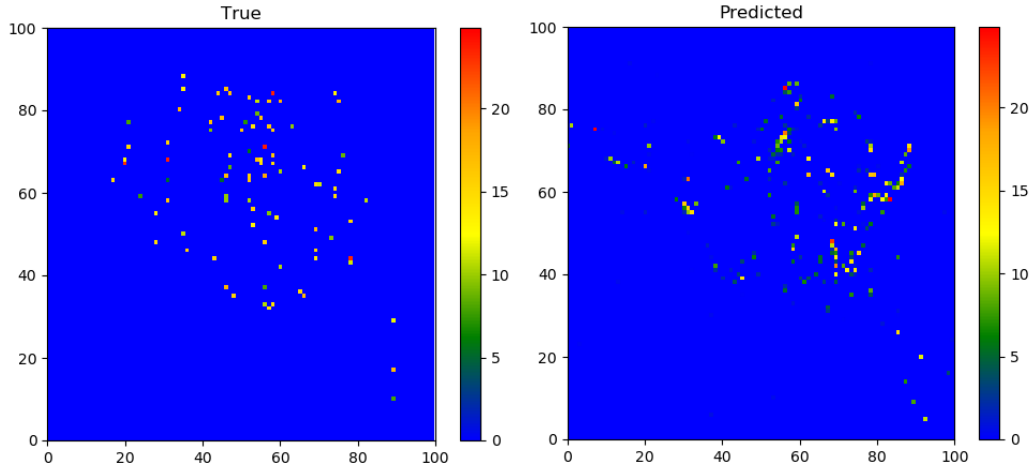


Figure 4.5: Validation example. True value after 40 minutes (left) and predicted value (right)

be the manifestation of overfitting, even if the validation loss curve seems to converge to the same value as the training loss. Several regularization methods have been tested, including L2 regularization, dropout, and Batch Normalization (see [16] for more details), but they significantly harm the overall performances. In particular, Batch Normalization leads to a model unable to provide any meaningful predictions. Unfortunately, the lack of evaluation metrics to assess the predictive performances of the model hampers the fine comparison of results from different models.

We provide now the results of the same basic encoder-decoder model but for predicting the density, rather than a measure computed with a complexity metric. The density in a given area is defined as the number of aircraft in this area at a given time step. Of course, the density values are normalized between 0 and 1 before being fed into the model. The model training and validation losses converges towards an MSE value of 3×10^{-4}, which is an order of magnitude below the convergence value when using the linear dynamical system metric. To measure the performances of the model, we consider the absolute errors between the predicted density and the true density over the validation set, when using the inference mode to compute the prediction. The validation set contains 271 examples, each example containing 10000 density values, we get a total of 2710000 values of absolute errors, among which 227841 errors are nonzero (which is expected since the airspace is mostly empty). We only plot these nonzero values in Figure 4.6 to have a more readable histogram. Most of the errors are located around zero, with significant spike at integer values. These spikes corresponds to the model overestimating or underestimating the number of aircraft in the corresponding areas. Most of the nonzero errors corresponds to the model predicting one aircraft in excess or one aircraft less than the true density. The mean of the nonzero errors is 0.9.

We use the same kind of analysis to compare several variations on the complexity metric and the dataset as presented in the methodology section. Figure 4.7a provides the histogram of nonzero absolute errors when the model is trained to predict the robust convergence values instead of the values computed with the linear dynamical system

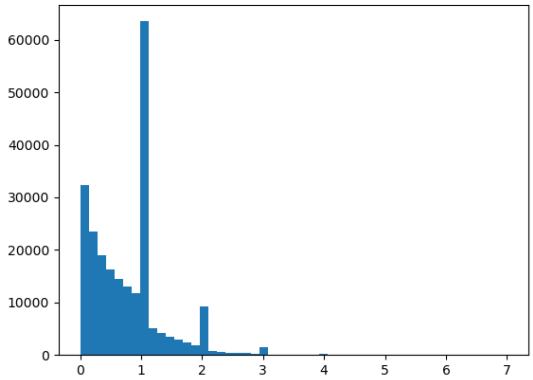
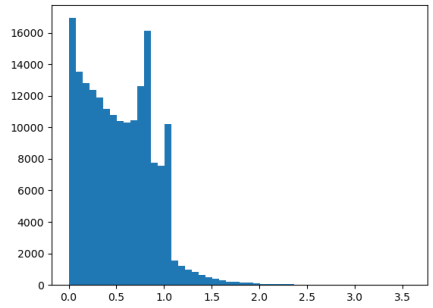


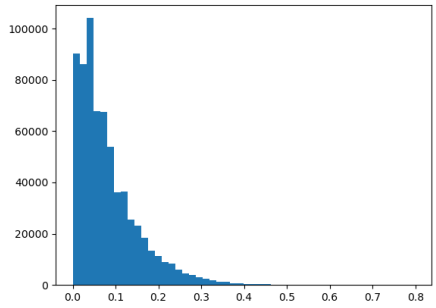
Figure 4.6: Distribution of nonzero absolute errors between prediction and truth over the validation set when predicting the density.

metric. Figure 4.7b shows the nonzero absolute errors when the output matrices $y[t]$ are smoothed with a Gaussian kernel (presented in chapter three). Figure 4.7c shows the histogram when using the linear dynamical system metric but with a time extension $m = 2$, while Figure 4.7d provides the histogram for the default model with $m = 10$ (the same model as used in the first paragraph of this section). For all four figures, the complexity values are normalized between 0 and 1, such that the maximum complexity value in the entire dataset is 1. The prediction are computed with the inference process. Let's first compare the robust convergence results with the default model. Both distributions of errors are very similar, in shape as well as in range. The default model has a slightly smaller average error with more errors below 0.1. The distributions are tendentially decreasing but with a significant amount of errors around 1 (meaning that the model predicts a maximal complexity where the complexity is zero, or the opposite), and a small proportion of errors are even above 1. Comparing the two variant of the linear dynamical system metric (with $m = 2$ or $m = 10$), the distributions once again feature a similar structure. However, with $m = 2$, the errors are spread across a smaller range and the mean is smaller. There is also a smaller number of nonzero errors overall. It is not known if this difference is significant, and if so, how to explain why this rather minor change in the computation of the complexity leads to such a variation in the performances of the predictive model. Finally, we consider the results for the Gaussian smoothing. Compared with the default model, the distribution of errors is decreasing more rapidly, the average error is five times smaller, but the total number of nonzero errors is higher. As already pointed out in section 3.2 for the sequence to vector model, the prediction task using the smoothed dataset seems to be easier for the encoder-decoder model.

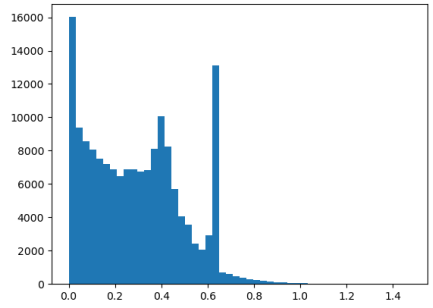
In this paragraph, we present the results with the binary classification model. We conduct a sensibility analysis on various hyperparameters, summarized in Table 4.1 and Table 4.2. The following are tested: the impact of the complexity metric (robust convergence or linear dynamical system), the impact of the complexity threshold η, the impact of applying a Gaussian smoothing on the complexity matrices $y[t]$ (before the thresholding with eta), and the impact of the choice of the input structure detailed at the end of the



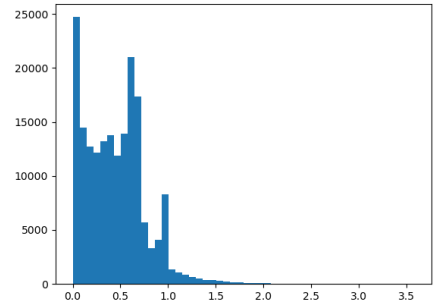
(a) with robust convergence metric
182408 values with mean = 0.5



(b) with Gaussian smoothing
678240 values with mean = 0.08



(c) with extension $m = 2$
161130 values with mean = 0.3



(d) with extension $m = 10$ (default model)
182982 values with mean = 0.4

Figure 4.7: Distribution of nonzero absolute errors between prediction and truth over the validation set with various settings

previous section. Since this is a classification task, we can use several metrics suited for classification, namely F1 score and MCC. To do so, we consider that the model predicted congestion in a given area when the output value is above 0.5. Let tp be the number of true positives, fn the number of false negatives, and fp the number of false positives. The proportion of predicted positives that are actually positive $\frac{tp}{tp+fp}$ is called the precision, while the proportion of positives that are predicted as positives $\frac{tp}{tp+fn}$ is called the recall. The F1 score is the harmonic mean of the precision and recall $F1 = \frac{tp}{tp+\frac{1}{2}(fp+fn)}$. It ranges from 0 when either the prediction or recall is 0, to 1 when both the precision and recall are equal to 1. The MCC (Matthews Correlation Coefficient) ranges from -1 (inverse correlation) to 1 (perfect correlation) and takes into account the true negatives tn. It is defines as $MCC = \frac{tp \cdot tn - fp \cdot fn}{\sqrt{(tp+fp)(tp+fn)(tn+fp)(tn+fn)}}$. In each cell of the tables, the top number is the F1 score while the bottom number is te MCC. In this experiment, the F1 score and the MCC seems to be fairly consistent wit each other. Let's first compare the two complexity tested metrics. With $\eta = 95\%$ and without Gaussian smoothing, the linear dynamical system metric leads to $MCC = 0.13$, while we get $MCC = 0.16$ with robust convergence. This tends to show that, with this task, using robust conver-

	50	75	95	99
without Gaussian smoothing	0.29	0.16	0.12	0.03
	0.28	0.15	0.13	0.05
with Gaussian smoothing	0.43	0.35	0.08	0.06
	0.37	0.33	0.15	0.07

Table 4.1: F1 score (top) and MCC (bottom) for Gaussian smoothing and choice of the complexity threshold (in %) using the linear dynamical system metric (with structure "by position")

	by position	by aircraft
without Gaussian smoothing	0.16	0.15
	0.16	0.15
with Gaussian smoothing	0.31	0.31
	0.34	0.33

Table 4.2: F1 score (top) and MCC (bottom) for Gaussian smoothing and input structure using the robust convergence metric (with complexity threshold = 95%)

gence leads to better performance. This is corroborate when comparing both metrics when using Gaussian smoothing, we get $MCC = 0.15$ with linear dynamical system and $MCC = 0.34$ with robust convergence. In every cases, the use of Gaussian smoothing improves the performance of the model (both tables). Concerning the threshold η, using smaller values seem to provide higher MCC and F1 score (Table 4.1). This may be explained by the fact that the more η is close to 50%, the more the dataset is balanced, and the task is easier. Finally, the comparison of the two proposed input structure (Table 4.2) does not highlight any difference in the performances of the model.

In this chapter, we implemented a different approach to tackle the task of predicting congested areas, by using encoder-decoder networks. We investigated several variations, including complexity metrics, and the reformulation of the task as a multi-label binary classification task. Even if the model seems able to learn the patterns of the congested areas, the performances are less promising when using the inference mode, with a large number of predictions with high errors values. Applying a Gaussian smoothing on the dataset seems to be the only parameter that significantly improve the performances. Concerning the classification task, several parameters have an impact over the performances, however all F1 score and MCC obtained during the tests remain quite small (below 0.5).

Chapter Five

Object detection model

5.1 Methodology

Until now, the general objective of predicting congestion in the next 40 minutes has been addressed directly with the following approach:

1. Compute a measure of complexity with a metric
2. Build a task such that the ground truth is the complexity value itself
3. Train a model that uses past and current information to predict this complexity value in the future, in each area of the airspace

In this chapter, we consider a different and more indirect approach. Remember that the objective of this work is to predict congested areas. The previous models achieved this goal by predicting directly the complexity as a regression task. However, what is more relevant in the point of view of a controller is to predict congested areas. The precise value of any complexity metrics in a given point is of little relevance once the location and shape of the future congested areas are known. From this perspective, the approach proposed in this chapter relies on redefining the target output of the model to emphasize the detection and characterization of congested areas, rather than complexity values. The first relevant information is the presence or absence of congested areas. This can be modelled with binary variables. If a congested area is predicted, the second relevant information is to provide a simplified shape, in particular the volume covered by the congested situation. To achieve both detection and characterization, we draw from the "You Only Look Once" (YOLO) algorithm presented in section 2.3. This algorithm was first intended to deal with object detection task, with minimal computation efforts. In this chapter, we will only consider the method used in the YOLO algorithm to define an object in the target outputs and not the model itself. In other words, we see congested areas as "objects" to be detected in an "image" of the airspace.

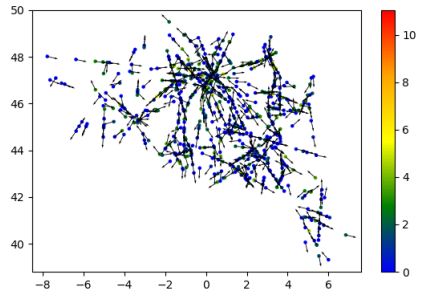
Let's now detail how to define a congested area. In our original dataset, we have for each time step t a set a aircraft in the airspace with a complexity value associated to each

aircraft (computed with the linear dynamical system metric). We now fixed a certain time step t and consider the aircraft present at t in the airspace (see Figure 5.1a). To be able to put these aircraft in an image, we only keep the horizontal position of each aircraft (the latitude and the longitude). A threshold is then defined for the complexity metric, and we remove all the aircraft which complexity value is below this threshold. We obtain a set of points corresponding to high complexity values (see Figure 5.1b). On these remaining points, we apply the DBSCAN algorithm (density-based spatial clustering of applications with noise [11]) such that we get a set of clusters containing high complexity points, and potentially several noise points. For each identified cluster, a box is defined to summarize the characteristics of the cluster (see Figure 5.1c). A box is defined by five parameters: the coordinates of the center point, the width and height of the box, and a rotation angle. These boxes are used for the same purpose than the bounding boxes of the YOLO algorithm. More precisely, a $N \times N$ grid is defined over the airspace. Each cell of this grid is a vector of $\mathbb{R}^6$ of the form $\begin{bmatrix} p & x & y & w & h & \phi \end{bmatrix}$ where p is the binary variable set to 1 if the corresponding cell contains the center point of a cluster and 0 otherwise. If $p = 1$, then x and y are the coordinates of the center point, w and h are the height and width of the box, and ϕ is the rotation angle of the box. If $p = 0$, all the other parameters are set to 0. Finally, we obtain for every time step t a target output $y[t]$ which is a $N \times N \times 6$ tensor (see Figure 5.1d). It should be noted that the clusters and their corresponding boxes modelling the congested areas are defined for each time step independently.

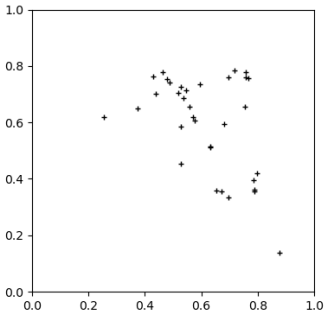
The inputs of the model remains the same as in the previous chapter, namely the Type II input defined as the concatenation of all aircraft states present in the airspace at time step t. We recall that the state of one aircraft is defined as a vector of dimension six containing the longitude, the latitude, the altitude, the ground speed, the heading, and the vertical speed.

We now present the models used to implement the object detection task. Two different architectures are tested. The first one is a sequence to sequence model with an encoder-decoder architecture. The encoder network is only composed of LSTM layers which transform the vectors of aircraft states into an encoding vector (the hidden state of the last LSTM layer computed after inputting the last time step). The decoder network is composed of LSTM layers followed by Dense layers. It implements teacher forcing (introduced in chapter four). Since it is a sequence to sequence model, the encoder-decoder model predicts the whole sequence of future congested areas, from the timestamp of the last input to 40 minutes in the future. The second model is a sequence to vector model, which only predicts the congested areas in 40 minutes (not the entire sequence), hence the task is easier. In this case, the model simply consists in several LSTM layers directly followed by several Dense layers.

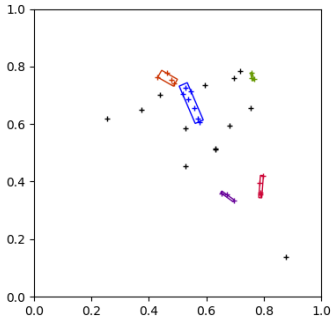
Considering the structure of the target outputs, we have to use a custom loss function. We use a simplified version of the YOLO loss function presented in section 2.3. This custom loss function is provided in Equation 5.1. We have that $\mathbf{1}_i = 1$ if the i-th grid cell contains the center point of a box, and $\mathbf{1}_i = 0$ otherwise. This means that if there is no congested area, the loss function is reduced to the square of the predicted variable $\hat{p}_i$ representing the probability that the corresponding cell contains a congested area as



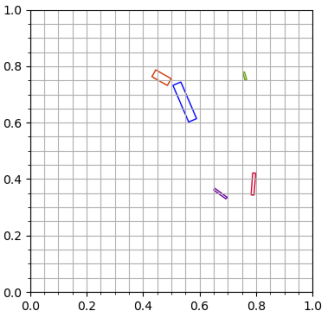
(a) Aircraft in the airspace at a given time step with the associated complexity value



(b) Remaining points after applying the threshold



(c) Five clusters have been identified by DBSCAN in this example (the black points are noise points)



(d) The target output is a $N \times N \times 6$ tensor

Figure 5.1: Definition of the target output

predicted by the model. The square roots used in the MSE error of the widths w and heights h aim at penalizing the errors on small boxes with respects to errors on large boxes. We introduce a parameter $\xi > 1$ in order to emphasise the error on the binary variable p when the box actually contains a congested area. The reason behind this hyperparameter is that most of the grid cells in the dataset do not contain any congested area, and so the true variable p is almost always 0. Without the hyperparameter ξ, the model is tempted to always predict small values to p even if it increases the error when the cell actually contains a congested area, since over the whole dataset, this increase in the loss is small with respect to the small errors when the cell is empty. This leads the model to never predict any congested areas. The hyperparameter ξ urges the model to improve its performances on the prediction of p.

$$\begin{aligned} L(y, \hat{y}) = & \sum_i \mathbf{1}_i (x_i - \hat{x}_i)^2 + \mathbf{1}_i (y_i - \hat{y}_i)^2 \\ + & \mathbf{1}_i (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + \mathbf{1}_i (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \\ + & \xi \mathbf{1}_i (p_i - \hat{p}_i)^2 + (1 - \mathbf{1}_i)(p_i - \hat{p}_i)^2 \end{aligned} \tag{5.1}$$

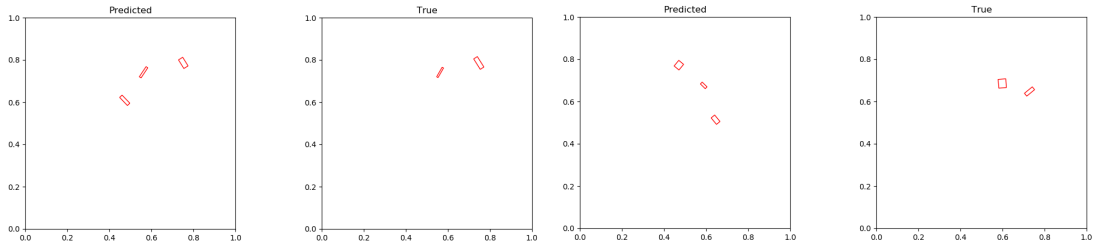


Figure 5.2: Predictions on the training set (both left figures) and on the validation set (both right figures)

5.2 Results

The results of the encoder-decoder model are presented first. To define the target output, we use a 20×20 grid ($N = 20$). The encoder network is composed of one layer of LSTM with hidden dimension 512. The decoder network is composed of one layer of LSTM with hidden dimension 512 followed by a Dense layer with output dimension 2400 and linear activation function. The input sequence has fixed length $T = 160$ as well as output sequences ($T_{pred} = 160$). The learning rate of the Adam algorithm is $\epsilon = 1 \times 10^{-3}$, the batch size is set at $bs = 128$ and the model is trained over 200 epochs. The hyperparameter ξ of the loss function is set to $\xi = 10$. The training and validation losses both converges towards almost the same value (2×10^{-3}). However, when the model is tested, it was unable to detect any congested areas (it always predicted $p = 0$ for all cells). This is why we implemented the sequence to vector model for which the task is assumed to be easier, since it only has to predict the congested areas at the 40th minute and not the entire sequence.

The sequence to vector model is composed of two LSTM layers both with hidden dimension 512 followed by two Dense layers with respective output dimensions 1024 and 1536. Their respective activation functions are ELU (Exponential Linear Unit) and linear (meaning no activation). We use a smaller grid with $N = 16$, and the hyperparameter of the loss function is set to $\xi = 5$. The model is trained over 150 epochs. All the other hyperparameters keep the same value. The training loss converges towards 8.5×10^{-3} while the validation loss converges towards 17.5×10^{-3}. If we consider only the variable p, and assume that the model detected a congested area when the predicted value of p is nonzero, we can compute the MCC (defined in the previous chapter). On the training set, we get $MCC = 0.61$, while it is only $MCC = 0.25$ on the validation set. The gap between the performances on the training and validation sets may indicate the need of regularization, as tested below. A sample of predictions with an example from the training set and another from the validation set are provided in Figure 5.2. On the training example, the model was able to detect two congested areas over the three actually present and managed to predict their locations and shapes with a reasonable accuracy. On the validation example, the model detected two congested areas over three but its predictions were less accurate.

Finally, we conduct a sensitivity analysis on the sequence to vector model. For these

	$\lambda = 10^{-4}$	$\lambda = 10^{-5}$
without batch norm	0.21	0.33
	0.033	0.033
with batch norm	0.06	0.001
	0.081	0.10

Table 5.1: MCC (top) and mean validation loss over the last 100 epochs (bottom) with input structure 'by position'

	$\lambda = 10^{-4}$	$\lambda = 10^{-5}$
without batch norm	0.26	0.29
	0.034	0.032
with batch norm	0.14	0.16
	0.058	0.36

Table 5.2: MCC (top) and mean validation loss over the last 100 epochs (bottom) with input structure 'by aircraft'

experiments, we add an third layer of LSTM with the same hidden dimension 512. The model is trained over 250 epochs. The other hyperparameters are identical. We investigate the use of regularization techniques (Batch Normalization and L2 regularization through the parameter λ) and the choice of input structure ('by position' or 'by aircraft' as defined in the previous chapter). To compare the performances, we use the MCC as well as the mean validation loss over the last 100 epochs of training. The results are summarized in the Tables 5.1 and 5.2.

As already found in the previous chapter with the encoder-decoder network, the input structure seems to have little impact on the performances of the model, as we can see by comparing the results of both tables. An L2 regularization with a small parameter λ may increase the performances since we obtain $MCC = 0.33$ or $MCC = 0.29$ with $\lambda = 10^{-5}$, to be compared with $MCC = 0.25$ obtained in the previous paragraph. If the value of λ is increased, the MCC decreases. However, Batch Normalization significantly harms the performances of the model (with MCCs very close to zero).

To conclude, the current performances of the object detection approach, as presented in this chapter, are promising when measured on the training set but far poorer on the validation set. A fine-tuning of the parameters of the models may improve these performances. Either way, this definition of the congested areas as clusters of high complexity is more meaningful from the operational perspective than a pure regression of the complexity as computed by a metric. This could also be possible to use the predictions of the former models from chapters three and four to define clusters of high complexity, without using a model which aims at directly detecting such clusters. A limitation of the current definition is that the congested areas are defined for one unique time step and hence do not have any temporal extension: there is no dynamics in the evolution of the clusters. This may be investigated in further extension of this approach.

Chapter Six

Trajectory prediction model

6.1 Methodology

In the three last chapters, we have trained RNN models to predict either the future value of a complexity metric or the position and shape of clusters of high complexity. In this chapter, we do not rely on RNN models to predict the complexity, but directly use a complexity metric to compute the complexity values. However, the complexity metric requires to know the aircraft state (in particular the aircraft position and speed) to compute the value of the complexity. The main problem is now to predict the aircraft state. In other words, we are considering a trajectory prediction problem. This is why we propose the following two-stage prediction method. During the first stage, a trained RNN model uses past information to predict the future trajectories (at least for the next 40 minutes). During the second stage, the complexity metric is computed using this predicted trajectory, providing the traffic congestion information.

This chapter is only focusing on the first stage of the method, namely the trajectory prediction task. Let's describe in detail the data available for the trajectory prediction. The model has access to three types of data: flight plan, true trajectory, and weather data. As introduced in section 2.5, we use three different flight plans from TLS to CDG respectively composed of a sequence of six, four, and eight 2D points. The flight plan also includes other information such as the initial mass or the aircraft type but only the sequence of 2D points is fed into the RNN models. Using these flight plans, wind data (presented in section 2.5), and the Aircraft Arithmetic Simulator, 862 different trajectories are generated. For the training of the models, the trajectory points are redefined with the following parameters: latitude, longitude, altitude, heading, northerly wind component, and easterly wind component. The wind components associated to a given trajectory point come from the point of the weather grid closest to the trajectory point.

When examining the shape of the trajectories generated by the simulator, we notice that for each flight plan the corresponding trajectories have all the exact same horizontal profile. This is due to the fact that the simulator does not adapt the generated trajectory

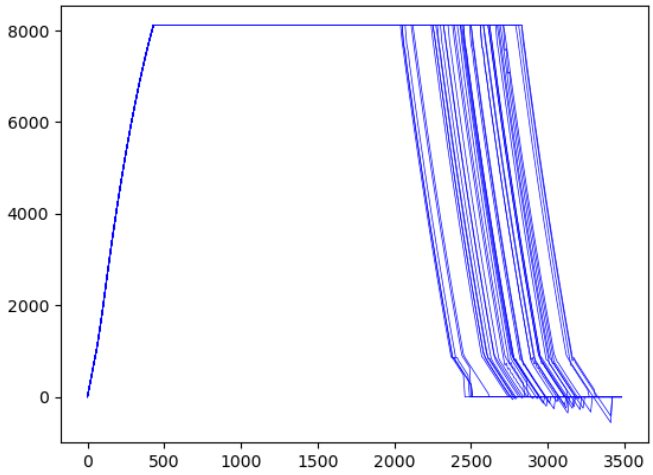


Figure 6.1: Altitude (meters) against time (seconds) for 43 trajectories from the training set

to the wind conditions, in particular by deviating from the flight plan. Regardless of the wind speed intensity and orientation (for example a strong headwind in one leg of the flight plan), the simulated trajectories always perfectly follow the flight plan. In this dataset, the only impact of the weather on the trajectories is the ground speed of the aircraft, and hence the duration of the flight (given the flight plan). More precisely, the main difference between trajectories generated from the same flight plan but with different weather conditions is the time of the top of descent (TOD), namely the time when the aircraft begins its descent. A subset of the simulated trajectories is plotted on Figure 6.1. This figure shows only the vertical profile of the trajectories, and more precisely the altitude with respect to the time. It is clear from this figure that the TOD is the only significant variation among the trajectories of the training set. This is a strong limitation of this dataset because the model is supposed to be able to predict the full 3D position of the aircraft at each time step, but due to the strong similarities of the training examples, the model is likely to have difficulties to learn correlations between input features and expected outputs. In fact, with this dataset, the only relevant parameters that are learnable by the model are the general shape of the trajectory (from the flight plan) and the time of the TOD (from the wind information).

We now present the model used for trajectory prediction. Taking inspiration from [27], we consider an encoder-decoder network as introduced in section 2.3. The encoder network is composed of several layers of LSTM, with the last hidden state of the upper layer used as encoding vector. The first LSTM layer takes one point of the flight plan as input (longitude and latitude), the number of time steps in the encoder network being equal to the length of the flight plan. The decoder network is made of several LSTM layers followed by several Dense layers. The teacher forcing technique is implemented such that the decoder network is fed with the true trajectory but delayed by one time step. At each time step, the first LSTM layer receives one trajectory point as input, each trajec-

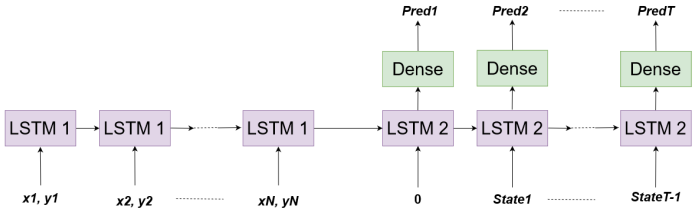


Figure 6.2: Encoder-decoder network for trajectory prediction.

tory point being of dimension six. Once propagated through the network, the last Dense layer directly outputs the prediction for the next trajectory point's position (hence the output has dimension three). The two wind components as well as the heading are not predicted. During the inference process, the true trajectory is not known. As already stated in chapters four and five, the last prediction of the decoder is used as the input for the next step. An illustration of the model is shown in Figure 6.2.

This model is a simplified version of the framework presented in [27]. We directly predict the position of the aircraft at the next time step, instead of predicting the parameters of a density function modelling the distribution of the next trajectory point. For this reason, we are not applying any beam search algorithm to select the most likely trajectory among a set of predicted possible trajectories. The loss function used is the Mean Squared Error between the predicted trajectory point and the true point, while in [27], the authors defined a cumulative log-likelihood for the whole trajectory with the parameters predicted by the decoder network in such a way that it is possible to rank the trajectories according to their likeliness. They also used the predicted parameters to filter the trajectory with an adaptive Kalman filter. All these improvements should be implemented in future versions of the model as discussed in the next section.

6.2 Results

For the training process, the dataset is normalized, with each variable being independently normalized between 0 and 1. Then, the dataset is split with 775 trajectories for the training set and 87 trajectories for the validation set.

For the experiments presented in this section, we have selected the following hyperparameters values. The encoder network is composed of two LSTM layers with hidden dimension 256. The decoder network is composed of two LSTM layers with hidden dimension 256 followed by 3 Dense layers of output dimensions 128, 64, and 3 respectively, with activation functions ELU for the first two Dense layers and linear activation for the last layer. The learning rate is set to $\epsilon = 10^{-4}$, the hyperparameters of the Adam algorithm are set to $\rho_1 = 0.9$ and $\rho_2 = 0.999$, with a batch size $bs = 64$. The model is trained over 200 epochs.

The MSE during training over the training set and the validation set respectively are plotted in Figure 6.3. We can see that both losses are converging towards a final MSE of 1.1×10^{-3}. This means that the model is able to learn from the training set and

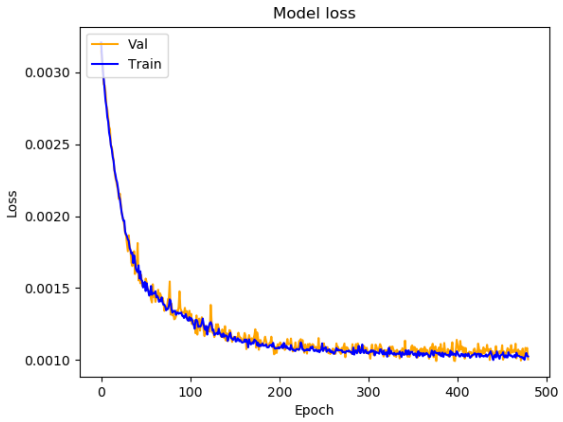


Figure 6.3: Training and validation losses during training.

to generalize efficiently to the validation set. However, the inference process achieves poorer results. Since at each time step of the inference process the decoder network uses its previous prediction as input, we observe a propagation of the errors, such that the predicted trajectory quickly diverges from the true trajectory. In contrast, when the decoder network is fed with the true trajectory, it is able to provide proper trajectory prediction.

This error propagation can be highlighted by changing the proportion of the true trajectory used as input during inference, as illustrated in Figure 6.4. In the left figure, the model receives 75% of the true trajectory as input (and uses the recursive mode for the last 25% of the trajectory), while in right figure, the model only receives 25% of the true trajectory. We can see that the model quickly deviates from the true trajectory as soon as it enters the recursive mode. Quantitatively, the MSE on the validation set during inference is 1.9×10^{-3} when using 75% of the true trajectory, but reaches 2.267×10^{-1} when using 25% of the true trajectory. As a comparison, the MSE is 1.1×10^{-3} when using 100% of the true trajectory (as during training). It seems that it is not possible to predict relevant trajectories with this approach without using a generative model (for example with a Gaussian mixture as proposed in [27]) and a beam search, and possibly smoothing the trajectory with a post-processing.

In the rest of the chapter, we will discuss possible paths of research aiming at using trajectory prediction to predict congested areas. We only sketch several ideas that have not been implemented during this internship but that could be the topic of future work. The simplest approach is to test basic LSTM models (without the encoder-decoder architecture), where a re-sampled flight plan is fed to LSTM layers which directly predict the deviation from the corresponding flight plan point according to the weather situation around this point.

Another approach is to address a different task where the objective is to predict only the time to some selected waypoints (for example the time of the TOD). Indeed, since the trajectories do not deviate from the flight plans, the main source of uncertainties for the prediction of congested areas is the deviations along the curvilinear abscissa.

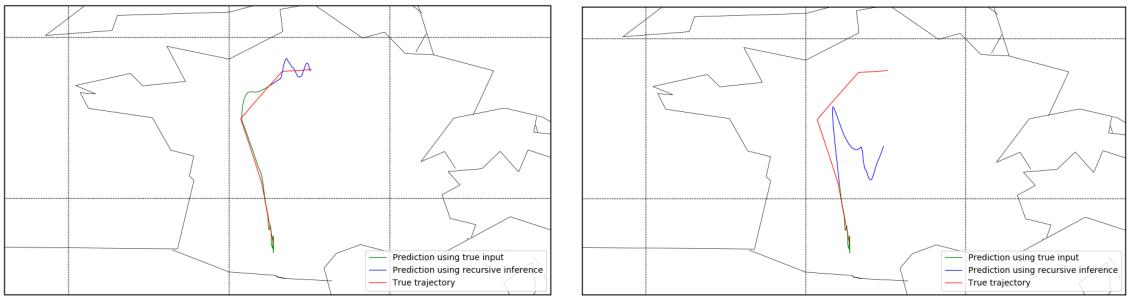


Figure 6.4: Examples of predicted trajectories with various proportion of true trajectory as inputs for the decoder network. In the left figure, 75% of the trajectory is generated using the true input (green curve) while it represents only 25% of the whole trajectory in the right figure. The red curve is the true trajectory which follows the flight plan.

These time predictions could be provided by deep learning techniques, for example using the information from radar tracks of other aircraft to predict useful parameters for the ground-based trajectory prediction of a given aircraft. Trajectory prediction can also be achieved with more classical methods which may be more suited for this task, since they are more stable and flexible. Deep learning techniques are more likely to overfit a specific dataset or a specific context and will have to be retrained when applied to a different airspace with different trajectories structures.

The bad quality of the trajectories predicted by the inference process precludes the application of the second stage of the method: using the predicted aircraft states to compute the complexity metric and provide a measure of the complexity in the next 40 minutes. Moreover, the computation of the metric requires to know all the trajectories in the considered area at the considered date, while the model presented here is only able to learn trajectories between a pair a chosen airports (namely TLS to CDG). A reasonable LSTM network (in terms of computation time and size of the dataset) seems unable to learn to predict the trajectories for arbitrary airports pairs. This is why, even with a successful LSTM model for one airports pair, we still have to train a model for each airports pair that are likely to cross our airspace. With one model for each airports pair, our framework may be computationally inefficient while lacking flexibility. A different approach could be to restrict our scope to a smaller area, and train an unique model to predict the trajectories from their entry in the sector until their exit.

Even if the results of the model described in this chapter do not meet the requirements of a proper trajectory prediction, the framework consisting in predicting congested areas from trajectory prediction seems to be the more promising. Hence, further experiments are needed with refined definitions of the goals of the prediction algorithms, in order to better address the operational context. More precisely, the operational need is not to provide a prediction of the complexity in every area of the airspace but rather to predict the formation of complex situations over well known waypoints that are prone to convergence of traffic.

Chapter Seven

Conclusion

In this report, various approaches have been presented to address the prediction of congested areas. We provide a literature review in chapter two on the topics of complexity metrics, recurrent neural networks, and several notions of deep learning, including applications to trajectory prediction. In chapter three, we have presented a simple RNN model using a sequence of aircraft states to predict the complexity of the air traffic in all areas of an airspace, in a time horizon of 40 minutes. A hyperparameters search was conducted to fine-tune their values. Then, we presented results showing the model's predictions with various inputs features. In chapter four, the same task is addressed with architectures drawn from Natural Language Processing, in particular encoder-decoder network with teacher forcing and 1D convolutional layers. Results are presented using several complexity metrics and various hyperparameters are tested. The encoder-decoder models are achieving better performances with respect to the sequence to vector model of chapter three, in particular when applying Gaussian smoothing on the output of the dataset before training the model. The reformulation of the objective as a multi-labeled binary classification is also investigated, but yielding less promising results.

In chapter five, a different approach is outlined where the model learns to detect and characterize congested areas defined as clusters of high complexity points, by taking inspiration from object detection algorithms. Some promising results are presented even if these models are not able to reach satisfactory performances. In chapter six, the prediction of congested areas is addressed in an indirect way, such that the main task of the deep learning models is trajectory prediction which is then used to compute a complexity metric. The results are only focusing on trajectory prediction with RNN models. The models presented in this chapter are unable to provide an appropriate trajectory prediction. These predicted aircraft states are not relevant enough to be used in the computation a complexity metric. Improvements of the RNN models for trajectory prediction have been proposed in chapter six but not yet implemented.

Future Work

Several potential improvements have been identified. To better address the needs of air traffic controllers, other approaches to the prediction of congested areas are possible. The main issue with hot spots detection is not their localization but rather the probability that they will appear in the near future at some well-known waypoints where the trajectories are converging. In this framework, the task of the prediction algorithm is to compute an estimated time of arrival over a set of pre-defined waypoints (as well as the altitude at which the aircraft will overfly the waypoints). With these predictions, it is possible to compute a measure of the complexity over every waypoint and hence predict the formation of hot spots. This approach is similar to define an AMAN-like algorithm over every selected waypoint. Another improvement is to build a model able to evaluate its own uncertainty with respect to its predictions. Deriving a probability or confidence score for each predicted congested areas can be achieved by propagating second-order moment through the network (rather than only propagating the first-order moment). This is possible by using the methods introduced in [10] for convolutional neural networks. The model should also be modified to take into account the future controllers' actions, since they impact the formation of congested areas. Concerning trajectory prediction with machine learning methods, another perspective is to use the radar tracks of several aircraft to predict unknown common parameters such that wind speed, and then hybridize these estimations with classical trajectory prediction algorithms in order to produce relevant predicted trajectories.

Finally, an evaluation by air traffic controllers from Flow Management Position is needed to assess the model's performances and to identify what level of performance is expected for the information provision from an operational perspective. Once evaluated, the prediction model can be used as an input to a mitigation model, which would be able to suggest actions on trajectories (changing heading, speed, or altitude) to prevent the formation of congested areas. This mitigation model could applied machine learning methods as well as more classical optimization techniques (such as metaheuristics, for example [18]) and conflict resolution algorithms. The final goal of such prediction and mitigation algorithm is to bridge the gap between the FMP's strategic planning and the controller's tactical monitoring. The objective would be to suggest upstream strategies to remove the hot spots that are too local and unpredictable at the strategic time frame, but still generate complex situations at the tactical level.

Glossary

ADAM	Adaptive Moment Estimation
AMAN	Arrival Manager
ATM	Air Traffic Management
BADA	Base of Aircraft Data
BPTT	BackPropagation Through Time
BRNN	Bidirectional Recurrent Neural Network
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
CNN	Convolutional Neural Networks
ELU	Exponential Linear Unit
ETA	Estimated Time of Arrival
FL	Flight Level
FMP	Flow Management Position
ISA	International Standard Atmosphere
GLM	Generalized Linear Model
GMM	Gaussian Mixture Models
GRU	Gated Recurrent Unit
HMM	Hidden Markov Model
LSTM	Long Short-Term Memory
MCC	Matthews Correlation Coefficient
MTCD	Medium Term Conflict Detection
MSE	Mean Squared Error
NLP	Natural Language Processing
NM	Nautical Mile
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
RVSM	Reduced Vertical Separation Minima
SGD	Stochastic Gradient Descent
TOD	Top Of Descent
TP	Trajectory Prediction
YOLO	You Only Look Once

Bibliography

[1] Samet Ayhan and Hanan Samet. Aircraft Trajectory Prediction Made Easy with Predictive Analytics. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 21–30, 2016.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *Proceedings of the 2015 International Conference on Learning Representations*, abs/1409.0473, 2015.

[3] Gano B. Chatterji. Short-Term Trajectory Prediction Methods. *Guidance, Navigation, and Control Conference and Exhibit*, 1999.

[4] Gano B. Chatterji and Banavar Sridhar. Neural network based air traffic controller workload prediction. *Proceedings of the American Control Conference*, 4:2620–2624, 1999.

[5] Gano B. Chatterji and Banavar Sridhar. Measures for air traffic controller workload prediction. *1st AIAA, Aircraft, Technology Integration, and Operations Forum*, 2001.

[6] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pages 1724–1734, 2014.

[7] Ramon Dalmau, Franck Ballerini, Herbert Naessens, Seddik Belkoura, and Sebastian Wangnick. Improving the predictability of take-off times with machine learning a case study for the maastricht upper area control centre area of responsibility. *SESAR Innovation Days*, 2019.

[8] A. M.P. de Leege, M. M. van Paassen, and M. Mulder. A machine learning approach to trajectory prediction. *AIAA Guidance, Navigation, and Control (GNC) Conference*, pages 1–14, 2013.

[9] Daniel Delahaye and Stéphane Puechmorel. Air traffic complexity based on dynamical systems. In *49th IEEE Conference on Decision and Control*, pages 2069–2074, 2010.

[10] Dimah Dera, Ghulam Rasool, and Nidhal Bouaynaya. Extended variational inference for propagating uncertainty in convolutional neural networks. *2019 IEEE 29th International Workshop on Machine Learning for Signal Processing (MLSP)*, 2019.

[11] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. of 2nd International Conference on Knowledge Discovery and*, pages 226–231, 1996.

[12] Pierre Flener, Justin Pearson, Magnus Ågren, Carlos Garcia-Avello, Mete Çeliktin, and Søren Dissing. Air-traffic complexity resolution in multi-sector planning. *Journal of Air Transport Management*, 13(6):323–328, 2007.

[13] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1027–1035, 2016.

[14] Adrian Garcia, Daniel Delahaye, and Manuel Soler. Air traffic complexity map based on linear dynamical systems. *Preprint*, 2020.

[15] Felix A Gers and Jürgen Schmidhuber. Recurrent nets that time and count. *Neural Networks, IJCNN*, 2:189–194, 2000.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[17] Klaus Greff, Rupesh K. Srivastava, Jan Koutnik, Bas R. Steunebrink, and Jurgen Schmidhuber. LSTM: A Search Space Odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232, 2017.

[18] Eulalia Hernández-Romero, Alfonso Valenzuela, Damián Rivas, and Daniel Delahaye. Metaheuristic approach to probabilistic aircraft conflict detection and resolution considering ensemble prediction systems. *SESAR Innovation Days*, 2019.

[19] Brian Hilburn. Cognitive Complexity in Air Traffic Control: A Literature Review. *Eurocontrol*, 12(13):93–129, 2004.

[20] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[21] Mariya A Ishutkina and Eric Feron. Describing Air Traffic Complexity Using Mathematical Programming. In *AIAA 5th Aviation, Technology, Integration, and Operations Conference*, 2005.

[22] Andrej Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks*, 2015. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`.

[23] Parimal Kopardekar and Sherri Magyarits. Dynamic density: Measuring and predicting sector complexity. *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, 1:1–9, 2002.

[24] I V Laudeman, S G Shelden, R Branstrom, and C L Brasil. Dynamic density: An air traffic management metric. Technical Report April 1998, NASA, Moffett Field, CA, 1998.

[25] Quoc V Le, Navdeep Jaitly, and Geoffrey E Hinton. A Simple Way to Initialize Recurrent Networks of Rectified Linear Units. *arXiv e-prints*, 2015.

[26] Keumjin Lee, Eric Feron, and Amy Pritchett. Air traffic complexity: An input-output approach. In *American Control Conference*, pages 474–479, 2007.

[27] Yulin Liu and Mark Hansen. Predicting Aircraft Trajectories : A Deep Generative Convolutional Recurrent Neural Networks Approach. *arXiv e-prints*, pages 1–24, 2018.

[28] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[29] John Lygerosl and Maria Prandini. Aircraft and Weather Models for Probabilistic Collision Avoidance in Air Traffic Control. *Proceedings of the 41st IEEE Conference on Decision and Control*, pages 2427–2432, 2002.

[30] Anthony J Masalonis, Michael B Callaham, and Craig R Wanke. Dynamic density and complexity metrics for realtime traffic flow management: Quantitative analysis of complexity indicators. *5th USA/Europe Air Traffic Management R & D Seminar, Budapest, Hungary*, 139, 2003.

[31] Stephane Mondoloni and Diana Liang. Airspace fractal dimension and applications. In *Fourth USA/EUROPE Air Traffic Management R& D Seminar*, pages 1–7, 2001.

[32] Bang Giang Nguyen. *Classification in functional spaces using the BV norm with applications to ophthalmologic images and air traffic complexity*. PhD thesis, Université de Toulouse 3 Paul Sabatier, 2014.

[33] Xavier Olive, Jeremy Grignard, Thomas Dubot, and Julie Saint-lot. Detecting controllers' actions in past mode s data by autoencoder-based anomaly detection. *SESAR Innovation Days 2018*, 2018.

[34] Yutian Pang, Houpu Yao, Jueming Hu, and Yongming Liu. A Recurrent Neural Network Approach for Aircraft Trajectory Prediction with Weather Features From Sherlock. *AIAA Aviation 2019 Forum*, pages 1–14, 2019.

[35] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *ICML (3)*, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 1310–1318. JMLR.org, 2013.

[36] Vu Pham, Theodore Bluche, Christopher Kermorvant, and Jerome Louradour. Dropout Improves Recurrent Neural Networks for Handwriting Recognition. In *Proceedings of International Conference on Frontiers in Handwriting Recognition, ICFHR*, volume 2014-Decem, pages 285–290, 2014.

[37] Rohit Prabhavalkar, Kanishka Rao, Tara N. Sainath, Bo Li, Leif Johnson, and Navdeep Jaitly. A Comparison of sequence-to-sequence models for speech recognition. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, 2017-August:939–943, 2017.

[38] Maria Prandini, Luigi Piroddi, Stéphane Puechmorel, and Silvie Luisa Brázdilová. Toward Air Traffic Complexity Assessment in New Generation Air Traffic Management Systems. *IEEE Transactions on Intelligent Transportation Systems*, 12(3):809–818, 2011.

[39] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016-December:779–788, 2016.

[40] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition. *arXiv e-prints*, 2014.

[41] George Saon, Gakuto Kurata, Tom Sercu, Kartik Audhkhasi, Samuel Thomas, Dimitrios Dimitriadis, Xiaodong Cui, Bhuvana Ramabhadran, Michael Picheny, Lynn Li Lim, Bergul Roomi, and Phil Hall. English conversational telephone speech recognition by humans and machines. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, 2017-August:132–136, 2017.

[42] Mike Schuster and Kuldip K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[43] Banavar Sridhar, Kapil S Sheth, and Shon Grabbe. Airspace Complexity and its Application in Air Traffic Management. In *2nd USA/EUROPE Air Traffic Management R&D Seminar*, pages 1–9, 1998.

[44] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 2014.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017.

[46] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. Sequence to sequence - Video to text. *Proceedings of the IEEE International Conference on Computer Vision*, 2015 International Conference on Computer Vision, ICCV 2015:4534–4542, 2015.

[47] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 07-12-June, pages 3156–3164, 2015.

[48] Hongyong Wang. Modeling Air Traffic Situation Complexity with a Dynamic Weighted Network Approach. *Journal of Advanced Transportation*, 2018, 2018.

[49] Hong Jie Wee, Sun Woh Lye, and Jean-philippe Pinheiro. A Spatial , Temporal Complexity Metric for Tactical Air Traffic Control. *The Journal of Navigation*, pages 1040–1054, 2018.

[50] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing [Review Article]. *IEEE Computational Intelligence Magazine*, 13(3):55–75, 2018.

[51] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent Neural Network Regularization. *arXiv e-prints*, pages 1–8, 2014.