University of Padova

Department of Mathematics "Tullio Levi-Civita"

Master Thesis in Computer Science

Studying and Improving Motion Indeterminacy Diffusion for Stochastic Trajectory Prediction

Supervisor
Prof. Lamberto Ballan
University of Padova

Co-supervisor
Sourav Das
University of Padova

Master Candidate
Enrico Buratto

Academic Year
2022-2023

”Now these points of data make a beautiful line and we’re out of beta, we’re releasing on time.”
- GLaDOS

Abstract

Human trajectory prediction is a fast growing subject in the Computer Vision field; the study of this problem is acquiring more and more importance due to its practical implications, among which autonomous driving and crowd surveillance are the most cited. Many different points of view are being analyzed in order to improve the precision and the correctness of the models; for instance, different data can be exploited to better understand how humans behave, and different mathematical frameworks can be used to improve the predictions.

One big acknowledgment in the field is that this is a challenging task due to the inherent indeterminacy human motion brings: free will and unpredictability are, in fact, intrinsic to human behavior. The authors of the paper on which this work is based try to solve this very issue: their idea, based on the concept of diffusion models, is to simulate this indeterminacy by applying random noise and subsequently try to learn how to remove it from the available data in order to obtain the most probable trajectories.

Their work, however, is based on an outdated version of diffusion models; many advancements have been made in the meantime, thus the idea for this work: we try to modify the model by applying the improvements, and thus try to improve its performance. In addition to this, we tried to expand the model by implementing a goal module, already developed and currently in improvement by the VIMP group at University of Padova, in order to include additional information and, therefore, to enhance the model capabilities.

Contents

Listing of figures

Listing of tables

Listing of acronyms

ADE . . . . . Average Displacement Error

CNN/CNNs . . . Convolutional Neural Network/s

CVAE/CVAEs . . Conditioned Variational Autoencoder/s

DDPM . . . . . Denoising Diffusion Probabilistic Models

FDE . . . . . Final Displacement Error

GAN/GANs . . . Generative Adversarial Network/s

GRU/GRUs . . . . Gated Recurrent Unit/s

IDDPM . . . . . Improved Denoising Diffusion Probabilistic Models

KDE-NLL . . . . . Kernel Density Estimation based Negative Log Likelihood

LSTM/LSTMs . Long Short-Term Memory network/s

MID . . . . . Motion Indeterminacy Diffusion

NLL . . . . . Negative Log Likelihood

RNN/RNNs . . . Recurrent Neural Network/s

SAR . . . . . Self-Attentive Recurrent network

VAE/VAEs . . . . . Variational Autoencoder/s

1 Introduction

Trajectory prediction, also known as trajectory forecasting, consists of the task of predicting the path an agent will take after traveling a certain amount of steps, which are known. The agent can be one or more different entities, depending on the domain under study. In the case of this work, we only focus on human agents, in particular pedestrians.

The practical implications of this research field are varied. One of the most obvious applications, and probably one of the most important on a usage basis, is autonomous driving: in the last decade a lot of research has been done on self-driving cars, and a lot of research is currently in development. In this domain, human trajectory prediction is obviously essential for cars to avoid accidents. Other kinds of trajectory forecasting are also needed since these vehicles strongly rely on the understanding of the surroundings to provide safe and smooth driving.
Another use for trajectory prediction is crowd surveillance and video surveillance in general: being able to predict the movement of an agent in the crowd can be very useful when talking about traffic control, accident prevention, and urban planning.
The last implementation we cite is socially-aware robots, *i.e.* autonomous machines that perform different tasks in a public environment. This topic is intertwined with Embodied AI, namely the research field where Artificial Intelligence and robotics cooperate to control physical appliances; in this context, predicting the future movements of a person is needed to avoid collisions between the robots and the person itself.

Apart from the history trajectory, which is used by almost every model ever developed, many other information can be exploited to improve the precision of the predictions; this task, in fact, is also referred to as a multi-modal problem just because of the many different types of data which can be used. For instance, social information can be used to exclude some generated trajectories since pedestrians are not meant to collide when they walk, or the model can take advantage of the aerial photos of the scene to state whether a path is plausible or not (*e.g.* it's unlikely for a person to cross a roundabout, while it's more probable for it to go around it).
Moreover, different computational and/or mathematical approaches have been taken over the years and are reported in the literature. These will be deepened in Chapters 2 and 3; at the moment, suffice to say that the most recurring computational approach is to use generative models to produce plausible trajectories and then to discriminate them on a precision metric basis.

Although the many different approaches made it possible to achieve good results, a well-known and hard-to-solve problem when talking about human motion prediction is the inherent indeterminacy it brings: free will and unpredictability are, in fact, intrinsic in human behavior, and it is an almost impossible problem to solve. This is where *Motion Indeterminacy Diffusion[1] (MID)* comes in: in this work, first published in March 2022, the authors apply the concept of diffusion models (of which the founding works are [2] and [3]) to the trajectory prediction task. The basic idea, which will be extensively explored in this work, is to apply random Gaussian noise to the spatial coordinates of the trajectories and then train a deep neural network to remove it; in this framework, the Gaussian noise represents the indeterminacy of human motion, and consequently, the neural network should be able to reduce this indeterminacy.
However, *MID* is based on an outdated version of diffusion models, namely *Denoising Diffusion Probabilistic Models[3] (DDPM)*; a new version of it, *Improved Denoising Diffusion Probabilistic Models[4] (IDDPM)*, has been released few months after the original one, and it brought several improvements both from the performance and the generation speed point of view. This is where the current work fits in: the main idea of this thesis, apart from studying and deepening this computer vision field, is to try to improve *MID* by applying the improvements from *IDDPM* in an iterative and incremental fashion; this way we could check, by extensive experimentation, whether an improvement could apply or not to this specific

domain.
Moreover, another contribution this thesis brings is the effort of merging two different models into one: we tried to expand the capabilities of *MID* by implementing the goal module coming from another work, namely *Goal-SAR[5]*, which is developed and currently in improvement by the VIMP group at University of Padova. This module consists in a neural network, which is used as a subroutine of the primary *MID* model, that tries to infer the final destination (viz. the goal) of the pedestrians.
Finally, as a side task, we decided to adapt the code from *MID* in order for it to work also in a long-term prediction framework and experiment with the resulting model in order to compare this approach with the actual state-of-the-art, which is *Y-Net[6]* and a discussion of it is given in Chapter 2.

Thesis structure

In this chapter, we saw a brief introduction to the trajectory prediction task, and a brief summary of the scope of this work has been given. The next chapters are organized as follows:

- In Chapter 2 we will analyze some relevant related works; the literature on the topic is vast, hence we report only the most important and on-topic works. In this chapter, we will describe the related papers, but the discussion on *Motion Indeterminacy Diffusion*, *Denoising Diffusion Probabilistic Models* and *Goal-SAR* is left for Chapter 3 due to their crucial importance for this work. Moreover, the analysis of *Improved Denoising Diffusion Probabilistic Models* is left for Chapter 5 due to its strong connection to the incremental methodology we followed for this work;

- In Chapter 3 a background for the problem is given: we will analyze the problem formulation in a detailed and formal way, the metrics used to assess the performances of the work, and all the background knowledge which will be needed to understand the subsequent chapters. In this section, we will also deepen the concept of diffusion models and we will study three out of four aforementioned papers, which are the founding basis for our work;

- In Chapter 4 a brief description of the datasets we used is provided;

- In Chapter 5 we will discuss the methodology we followed for this work. In particular, in this chapter, we will explain and discuss what this work consisted of, including the ideas and the intuitions behind the work, and the reasons that led us to make certain choices. Due to the incremental and iterative methodology which has been followed,

we will analyze the improvements step by step by analyzing the improved version of *Denoising Diffusion Probabilistic Models*;

- In Chapter 6 are listed the experiments we performed both on the existing project and on the developments we carried. In this section we will also report the results we got from these experiments;
- In Chapter 7 we will provide a short summary of the work and we will discuss the future work that can be done.

2
Related works

In this chapter we will analyze some works which are related to the current thesis. Since the literature on diffusion models applied to the trajectory prediction task consists, at the best of our knowledge, of the already cited *MID* only, this chapter only describes the state-of-the-art approaches over the last few years; in doing so, we decided to partition it in three main frameworks for trajectory forecasting: socio-temporal information, multimodality and goal-driven prediction.

2.1 Socio-temporal information

As outlined in the introduction, many different types of data can be exploited to create and/or improve a model for trajectory prediction; a first family of models is the one that tries to exploit information coming from social cues with respect to time. In this section, two crucial socio-temporal models are explored: *Social Forces* and *Social-LSTM*.

2.1.1 Social Forces

Social forces, short for *Social force model for pedestrian dynamics* [7], can be considered as a foundational work when talking of socio-temporal approaches to human trajectory forecasting. The main idea, inspired by physics concepts, is that the motion of pedestrians is strongly influenced by what the authors call *social forces*; these forces are not to be intended

as actual forces, but as an indicator of the stimulus of individuals to perform certain movements during walking, as the paper affirms. They define these forces as vectorial quantities of a pedestrian in relation to the time; these forces model the change of the preferred velocity of the pedestrian.

This work classifies these forces into three categories: the force which gives the acceleration to achieve the preferred velocity, the repulsion forces, and the attractive forces; all of these are then summed to get a final formula for human motion in a crowded scene. The acceleration is defined as the force needed to reach the preferred velocity; the path is defined to have the shape of a polygon because it is assumed that the pedestrian would try to take the shortest way. In this force, the authors also consider a relaxation time, which is the deceleration due to the approaching the goal. The repulsion forces are, as the name suggests, forces which push pedestrians away from each other; these are due to the private sphere of each person, which the authors suppose everyone has. Contrariwise, attractive forces are forces which push people close to each other; this could come, for instance, from meeting a friend along the path.

Even though the results from this work have been experimented with computer simulations, it cannot be considered as an actual state-of-the-art model due to its oldness; it constitutes, however, an important basis for the subsequent research on the topic.

2.1.2 Social-LSTM

Social-LSTM, short for *Social LSTM: Human Trajectory Prediction in Crowded Spaces* [8], is another important building block of the social-inspired models for trajectory forecasting. As opposed to *Social Forces*, the main idea of this work is to model human movement by learning it with Long Short-Term Memory networks (LSTMs, which will be deepened in Section 3.2).

For the authors, tailor-made functions like *Social Forces* are limited in representing the motion, as they tend to fail to generalize in complex crowded settings; this is because they tend to model interactions for a specific setting instead of learning interaction cues using real data. Moreover, the authors claim that this kind of models tend to consider only closely interacting people, but fails to consider possible future interactions. Therefore, they try to

address these problems by using LSTMs; in addition to these, they also add what they call a "social pooling layer", thus creating the real *Social-LSTM*.

The general architecture of this model is quite straightforward: since each person moves differently, the authors decided to assign an LSTM to any of the pedestrians in the scene. Moreover, since the LSTMs are not able to grasp information about the other agents, they decided to connect the neighboring ones using the aforementioned social pooling layers. At each timestep, then, the recurrent networks receive the pooled information from the neighboring ones. This way, the model is able to learn the interactions between pedestrians. In order to generate future trajectories, the hidden state t from the pooling layer at time t is used as a probability distribution from which the next future step can be sampled. The set of all the samples coincides with the final prediction.

2.2 Multimodality

Another much-discussed topic in trajectory forecasting is multimodality; in the literature, this can be referred to as either a perspective where many different information are considered, or just many highly-distinct future behaviors are considered. *Trajectron* is a representative of the latter, while its improved version *Trajectron++* is representative of the former. These two works are here briefly described, as their concepts and parts of code are also used in *MID*, as we will better see in Chapter 3.

2.2.1 Trajectron

Trajectron, formally *The Trajectron: Probabilistic Multi-Agent Trajectory Modeling With Dynamic Spatiotemporal Graphs* [9], is a 2018 work from Stanford University, which is one of the leading organizations in the trajectory prediction field; at the time, their approach resulted to be the state-of-the-art, and it gave birth to one of the most influential works when talking of human trajectory forecasting: *Trajectron++*.

The main idea of the authors is to present a graph-structured model that considers all the agents in a scene, as opposed to the previous works which used to consider just one trajectory at a time; as seen, social interactions have already been considered and studied since *Social Forces*, but these works never considered the scene as a whole, unique graph. In order

to model all these trajectories, the authors proposed a combination of recurrent sequence modeling and variational generative modeling to produce a probability distribution for each of them. This way, they claim to address the multimodality of human motion: instead of having just one future trajectory per agent, we would have a probability distribution, which brings even more representational power. This concept is also one of the fundamental ideas of *MID*, as we will analyze deeply in the next chapters.

The architecture of this model is quite complicated, and it is not this thesis' goal to explain it; however, the general idea is here given. *Trajectron* is composed of three elements which work together: a variational generative model called CVAE (Conditioned Variational Autoencoder), a recurrent model, namely an LSTM, and a spatio-temporal structure which addresses the multimodality; the first two components' concepts are analyzed in Chapter 3, while the latter is modeled as an undirected graph where the agents are represented by nodes and the edges correspond to the spatial proximity between them, as *Social-LSTM* does. Instead of proper positions, *Trajectron* considers the velocity of humans; this can be easily computed by differentiating the space with regard to time. The model then tries to learn the probability density function of each agent by optimizing the lower bound of the expectation of the sum of all the agent's probability distributions.

2.2.2 Trajectron++

An advancement of *Trajectron* is *Trajectron++*, a model coming from a 2021 work by the same researchers at Stanford University called *Trajectron++: Dynamically-Feasible Trajectory Forecasting With Heterogeneous Data* [10]. The idea here is that human motion is not influenced by just one factor, such as social interactions or the semantics of the scenes the pedestrians are moving inside, but by many of them: a factor cannot be considered as standalone, but all the information should contribute to the model. As a result, they propose then an improvement of the original paper with some, crucial additions.

As for its predecessor, this work relies on a graph-structured recurrent model which addresses the multimodality of human motion. However, as opposed to the original approach, different agent semantics are considered: in a scene, there are not only pedestrians but also what they call *ego-agents*, *i.e.* non-human agents with a human-like behavior; examples of these are cars, bicycle, busses, which are effectively controlled by persons. All the agents are

represented as nodes in a spatio-temporal graph, and their interactions are modeled as the edges of the graph as *Trajectron* did; however, differently from the earliest, the graph is directed. The reason for this, as the authors claim, is that it can represent a more general set of behaviors and interactions; for instance, a car driver has a worse perception of what is happening behind them. Moreover, each node in the graph has a semantic class, which can be pedestrian, bus, car, or other types of agents. Finally, as mentioned, semantic data about the scene is given to the model in order to grasp the context in which the agents are moving, thus improving the likelihood of the trajectories.

The general idea of *Trajectron++*'s functioning can be summed up in steps, most of which can be done in parallel and/or in a different order. The first step consists in creating the directed graph; then, the model encodes the nodes' current states, histories, and interactions by feeding observed history, current and previous states, and the semantic class to an LSTM network. In parallel, the agent interactions are encoded by aggregating the neighboring agents of the same class, feeding this aggregation to another LSTM, and then by applying an additive attention method (similar to what we will describe in Section 3.3.2) to obtain one representation vector, which the authors call *influence vector*. The fourth step is to incorporate heterogeneous data: in this project specifically, the authors encode a semantic map by using a Convolutional Neural Network (CNN); the outcome of this process is another representation vector. It is important to notice that further semantic data can be used by just extending this component with another network and including its output as another representation vector. Finally, ego-agent motion plans are encoded by using a bi-directional Recurrent Neural Network (RNN). All this information is fed to a Gated Recurrent Unit (GRU), a special case of LSTM which we will cover in Section 3.2, that outputs the parameters of a bivariate Gaussian distribution; these parameters model control actions such as acceleration and steering rate, and are therefore integrated numerically to obtain the final predicted trajectories.

2.3 Goal-driven prediction

A last approach we analyze is the goal-driven prediction; in this context, the goal, *i.e.* the final or intermediate destinations, of an agent is exploited to compute more realistic trajectories. Here, it is important to notice that with *goal-driven*, we mean that we use the goal information in addition to other information, as we will see in the upcoming sections.

2.3.1 PECNet

PECNet, short for *Predicted Endpoint Conditioned Network* [11], is one of the most influential works among the goal-driven category; in this work, the authors propose an inference method based on different intermediate and a final goal for (only) pedestrian trajectories. The reason behind this idea comes from three assumptions, which the authors report: first, the pedestrians should have some idea of where they want to go, and this holds also for intermediate destinations. Second, the pedestrians plan a trajectory according to the scene semantics; third, the trajectories can be modified while moving due to social interactions.

In agreement with these considerations, the authors propose then a decomposition of the problem into two smaller sub-problems: the first is to estimate a probability distribution which models the pedestrians' endpoints and then sample the most probable (intermediate and final) endpoints; the second is to generate the actual trajectories by taking into account the past trajectories, the scene semantics and the other pedestrians in the scene. With regards to the latter sub-problem, it is important to notice that the trajectory of a pedestrian is not conditioned only on the information about its neighbors, but also on everybody's estimated endpoints.

Practically speaking, the first problem is addressed by what they call an Endpoint VAE; this is a CVAE which encodes the history path of each pedestrian in the scene and of each ground truth endpoint into a latent variable, which produces the mean and the variance of a Gaussian distribution. At test time, possible endpoints are sampled from the linked Gaussian distribution; these are then fed to the decoder, together with the past motion history. The decoder generates plausible endpoints, which are in turn fed to a social pooling network, together with the encoded representation of the past trajectory. This network is composed of a given amount of rounds of social pooling, where at each round the representation of the previous one is updated according to a non-local attention mechanism (which is defined in [11] and not reported). At the end of this process, the remaining output consists in a variable that contains the information about past and future motion of all the agents in the scene.

2.3.2 Goal-GAN

Goal-GAN, whose name comes from *Goal-GAN: Multimodal Trajectory Prediction Based on Goal Position Estimation* [12], is another milestone when talking of goal-driven trajectory

prediction. In this framework, the authors model the task in a two-stage process: the former process is the goal estimation, which aims to predict the most plausible goals of the agent; the latter one is a routing module which estimates likely paths between one endpoint and another. Also in this case, goals are intended as both intermediate and final.

Considering that, the authors propose the aforementioned two-step procedure. Firstly, the possible goals of each pedestrian are computed by taking into account the dynamics of the pedestrian (*i.e.* its velocity and its social interactions) and the information about the scene, which comes from aerial RGB images of the scene itself. Secondly, this information is used to produce trajectories that pass through the predicted endpoints; this way, the authors claim, the multimodality of human motion is taken into account.

Going into some details of the model, this is composed of three components, which again work together to predict the final plausible trajectories. The first component is the Motion Encoder, which extracts the information about pedestrians' speeds and directions; then, it uses an LSTM to encode the trajectories, and the hidden state of this is used by the other modules to both predict the goal and produce the future trajectory for each pedestrian. The second component is the Goal Module, which predicts the goals for a given pedestrian given the features extracted by the precedent module and the information about the scene. To this end, an Encoder-Decoder network consisting of a CNN and an LSTM is used to extract information from an aerial RGB photo of the scene. This information is then concatenated with the dynamic information and fed to the decoder, which outputs a probability distribution of the predicted goals. The third and final model is the Routing Module, which generates the actual trajectories passing through the predicted goals. In order to do so, this module is made up of an LSTM network which uses a visual soft attention network to output feasible trajectories.

2.3.3 Y-Net

As a last example of goal-driven prediction, we analyze a paper which is, at the time of writing this thesis, competing for the state-of-the-art award. This work is called *From Goals, Waypoints & Paths To Long Term Human Trajectory Forecasting* [6], but it is often referred to as simply *Y-Net*. As for the authors of *Goal-GAN*, the foundational idea for this work is that uncertainty in future human trajectories is given by two sources: the ones which are known

to the agent such as the long-term goals, called by the authors *epistemic*, and the ones which are given by the surrounding environment such as the scene context, other agents and the always present randomness in human decisions, called *aleatoric*.

From a concrete point of view, the model uses two information sources: the RGB image of the scene and the past trajectory history. They use this information to produce a probability distribution that represents the long-term goals of the agents; these address epistemic uncertainty. Moreover, they also compute probability distributions of waypoints, which are the equivalent of the intermediate goals of *Goal-GAN*, in order to address the aleatoric uncertainty and reduce it.

Before continuing with a brief model explanation, it is of crucial importance to notice that *Y-Net* authors introduce a new framework setting in the trajectory prediction field: they are the first, in fact, to propose a long-term prediction alongside the traditional prediction, which can be named short-term. This will be better analyzed in Section 3.1; at the moment, suffice to say that the ratio between the future predictions and the history path in a long-term prediction setting is higher than in a short-term one. For this reason, it is also important to notice that the number of waypoints in the short-term scenario is zero, while they are present in the long-term one.

Talking about the model structure, this is composed of several sub-networks, which are mainly based on U-Net [13]. A first sub-network deals with image processing: the RGB image containing the aerial view of the scene is processed with a segmentation network (in this case, a U-Net) in order to obtain a segmentation map comprising a certain amount of classes. In parallel, the past motion history is converted to a heatmap with the same dimensions as the segmentation map. As opposed to previous works, the authors decided to adopt this technique (which they call *trajectory-on-scene heatmap*) in order to maintain the alignment with the scene; this way, they claim, there are no information losses due to the dimensions shrinking caused by CNNs. The two maps are then concatenated and processed with a U-Net encoder comprised of a certain amount of blocks M; this reduces the spatial dimensions by applying max-pooling operations and then passes each intermediate tensor to two decoders: a *goal decoder* and a *trajectory decoder*. The former consists of the expansion arm of the U-Net, and it produces a non-parametric probability distribution of the final and intermediate goals; the latter is still modeled as the expansion arm of the U-Net but, as opposed to the

first decoder, it is conditioned on the goal and the waypoints sampled for the probability distribution of the first one, the scene information and the history trajectory. The output of the trajectory decoder is then a stack of probability maps with two dimensions equal to the scene dimensions, and the number of channels equal to the number of future timesteps; a sampling is finally performed from each channel in order to obtain the final trajectory.

3 Background

3.1 Trajectory prediction

As already mentioned in Chapter 1 and discussed in Chapter 2, trajectory prediction is the task of forecasting the future trajectory of an agent based on a certain amount of known steps it took. In this section, we explore this concept in a formal way, reporting also the standard methodologies and the metrics which are commonly used to evaluate the performances of a given model.

3.1.1 Problem formulation

The one of trajectory prediction is a time series and regression task: the former means that the data points have a time order which must be considered; the latter means that it is composed of one or more response variables which depend on a set of independent variables. The considered task is composed of essentially three-dimensional data (to which other information is added depending on the implementation of the model that solves the problem, as we saw in Chapter 2); this crucial data is the time t and spatial information (x, y).
In practice, the datasets are composed by scenes, which are aerial scenes of real-existing places, in which some pedestrians are recorded while moving: their coordinates for a certain amount of timesteps T_{obs} are known, and we must predict their coordinates for another amount of time steps T_{pred}. The former is also known as *history*, while the latter is known as *future*;

the standard framework in literature reports these to be $3.2s$ for history and $4.8s$ for future [8],[14]. Usually, this translates into an amount of $T_{obs} = 8$ and $T_{pred} = 12$ time steps; the sampling rate, in fact, is commonly fixed at one sample each $0.4s$.
In addition to this framework, which can be called short-term prediction, a new one is recently being explored; this is called long-term prediction, and its first appearance is in [6]. In this case, we use $5s$ as history time and $30s$ as future time, with $T_{obs} = 5$ and $T_{pred} = 30$ time steps since a step is sampled each second. This scenario is also analyzed in [5].

3.1.2 Metrics

In order to assess the precision of the trajectories a given model generates, two standard metrics are used: Average Displacement Error (*ADE*) and Final Displacement Error (*FDE*); these were originally defined in [15], and from this work they became the gold standard for trajectory prediction evaluation.
In addition to these, in this work we also consider Kernel Density Estimate based Negative Log Likelihood (*KDE-NLL*) as firstly defined in [9]; this metric, in fact, can grasp the changes in the model output's likelihood, and this will become important when we will talk about the model improvements in Chapters 5 and 6.
Finally, at the time of writing, other metrics are being explored in contrast with the more traditional ones, as [16] suggests. These are analyzed in the last paragraph of this section for completeness but were not used in this work due to the lack of sufficient literature.

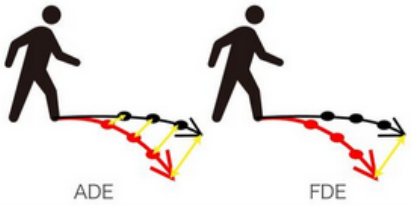

Figure 3.1: ADE and FDE metrics visualized. Source: https://speakerdeck.com/himidev/.

Average Displacement Error

The Average Displacement Error is defined as the ℓ^2-norm (also known as euclidean distance) between the predicted and the ground truth position at each timestep. Intuitively, this metric represents the general error between the entirety of the predicted trajectory and the real one; therefore, the lower the metric is, the closest to the ground truth the predicted trajectory is.

It is defined by the following formula:

$$ADE = \frac{\sum_{i=1}^{n} \sum_{T_{obs}}^{T_{pred}} \left[\left(\hat{x}_i^t - x_i^t \right)^2 + \left(\hat{y}_i^t - y_i^t \right)^2 \right]}{n \cdot (T_{pred} - T_{obs})} \tag{3.1}$$

where n is the number of pedestrians in the analyzed scene, $\hat{x}_i^t$ and $\hat{y}_i^t$ are the (x, y) coordinates of the predicted trajectory at timestep t and x_i^t and y_i^t are the (x, y) coordinates of the ground truth trajectory at timestep t.

The common methodology for computing the *ADE* of a stochastic and generative model output is to sample 20 trajectories, calculate the metric for all 20 of them and then take the best one (also called *BoN* - Best of N); therefore, a more correct notation for *ADE* is $min_{20}ADE$.

Final Displacement Error

The Final Displacement Error is defined as the ℓ^2-norm between the predicted and the ground truth position at the last timestep. Intuitively, this metric represents how close the last predicted step is to the last real step; thus, for the same reason of *ADE*, the lower the metric is the closest to the truth the final goal is.

It is defined by the following formula:

$$FDE = \frac{\sum_{i=1}^{n} \sqrt{\left(\hat{x}_i^{T_{pred}} - x_i^{T_{pred}} \right)^2 + \left(\hat{y}_i^{T_{pred}} - y_i^{T_{pred}} \right)^2}}{n} \tag{3.2}$$

where n is the number of pedestrians in the analyzed scene, $\hat{x}_i^{T_{pred}}$ and $\hat{y}_i^{T_{pred}}$ are the (x, y) coordinates of the last predicted point and $x_i^{T_{pred}}$ and $y_i^{T_{pred}}$ are the (x, y) coordinates of the last real point.

As for *ADE*, the common methodology is to consider the best result out of 20 generated samples; therefore, a more correct notation is $min_{20}FDE$.

KDE-based Negative Log Likelihood

The *KDE-NLL* metric consists in the mean of the negative log-likelihood between the ground truth and a distribution created by fitting a kernel density estimate on the predicted samples. The definition of the latter goes beyond the scope of this work, but in short it is a non-parametric method to estimate the probability density function of a random variable based on kernels (*i.e.* non-negative functions like, for instance, uniform or normal functions). Intuitively, this metric represents how far are, cumulatively, all the predicted trajectories from the ground truth, or in other words how concentrated the predicted trajectories are in the same area as the ground truth ones.

This metric does not have a standard notation formula, but can be summarized by the following:

$$-\sum_{i=1}^{B}\sum_{t=1}^{T}\frac{\log pdf\left[kde\left(\left(\hat{x}_i^t,\hat{y}_i^t\right),\left(x_i^t,y_i^t\right)\right)\right]}{B\cdot T} \tag{3.3}$$

where B and T are respectively the number of batches and the number of timesteps, pdf is the probability distribution function of the first component (*i.e.* kde) on the points of the second component, kde is the Gaussian kernel density estimation function on the points of the predicted trajectories $\left(\hat{x}_i^t,\hat{y}_i^t\right)$ and x_i^t, y_i^t are the coordinates of the ground truth trajectories. An example of implementation of kde is the one from SciPy [17].

New advancements in metrics

As mentioned at the beginning of this section, some criticism has been moved towards *ADE* and *FDE* metrics. In particular, the authors of [16] report that both the two metrics are not sensitive to distribution shifts; that is to say, they ignore the set of generated samples, and this can lead to the extreme to a model which might generate just one sample close to the ground truth while the others are way off it. This was apparently a known problem, thus the authors of [9] introduced the *KDE-NLL* metric we reported above; however, this measurement is also disputed, since it appears to be susceptible to kernel function changes.

To work around these problems, two more metrics are introduced: Average Mahalanobis Distance (*AMD*) and Average Maximum Eigenvalue (*AMV*). In general, the first evaluates how close the distribution computed by the generated samples is to the ground truth, while the second evaluates the spread of the prediction probability distribution.

While these metrics are promising for a better model evaluation, however, they are still new at the moment of this writing, and a scarce amount of papers which use them has been published; therefore, these are not used in this work.

3.2 Recurrent Neural Networks, LSTMs and GRUs

Recurrent Neural Networks, also known as RNNs [18], [19], is a family of neural network models where the output from a node can influence the subsequent input to the same node. Even though they have been directly used for trajectory prediction in past approaches, they are not strictly within the scope of this work. However, a discussion on the topic is needed to better understand attention mechanisms, which are in turn needed to explain *Goal-SAR*; therefore, a generic description of them is here given.

As just mentioned, RNNs allow their nodes to create a cycle, unlike feed-forward neural networks where each node of each layer is fully connected with the nodes of the subsequent layer; a representation of this is given in Figure 3.2.

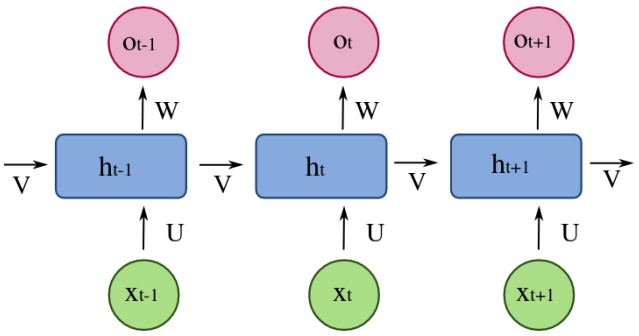

Figure 3.2: A portion of a recurrent neural network. Source: wikimedia.org.

At each timestep, the activation and the output are defined by the following mathematical representation:

$$\begin{aligned} h_t &= \sigma(\mathbf{V}_h x_t + \mathbf{U}_h h_{t-1} + \mathbf{b}_h) \\ o_t &= (\mathbf{W}_o h_t + \mathbf{b}_o) \end{aligned} \tag{3.4}$$

where $\mathbf{V}, \mathbf{W}, \mathbf{U}$ are weight matrices, σ is a non-linear activation function and $\mathbf{b}_h, \mathbf{b}_o$ are bias terms. The peculiarity of RNNs consists precisely in the presence of the $\mathbf{V}$ matrix: this set of parameters, in fact, connects a hidden layer of the network with the previous one.

RNNs are mainly used for sequence-like data such as natural language and time-series problems; therefore, it is quite obvious that they can be adapted for a task such as trajectory prediction. An advancement in RNNs are LSTMs [20], namely Long Short-Term Memory networks, depicted in Figure 3.3.

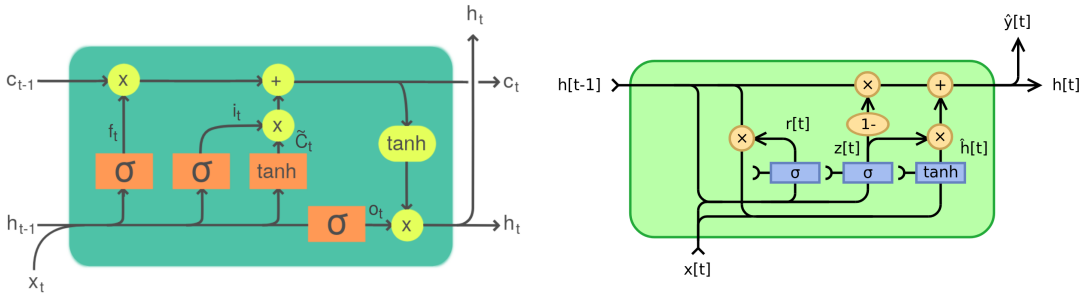

Figure 3.3: A cell from an LSTM (left) and from a GRU (right). Source: wikimedia.org.

The main purpose of these networks is to provide a short-term memory that can last several timesteps; this way, it enables the network to learn long-term temporal dependencies. In more formal terms, they partially solve the vanishing gradient problem[1] by allowing the gradients to propagate without being changed. In order to do so, LSTMs are composed by four main components: a cell state and three gates, namely input, output and forget gates; the gates set how the information flows during the training, while the cell state enables to remember values during the time.

When the data flows, the first step the network takes is to possibly choose which part of the information contained in the cell state to discard; in order to do so it activates the forget gate, *i.e.* the leftmost sigmoid σ in the figure, observing to the data $\mathbf{x}_t$ and the previous state h_{t-1}. Then, it chooses the information to store in the cell state by using the input gate, which decides which values are to be updated and is represented by the second leftmost sigmoid,

[1] A problem which occurs during backpropagation: being a computer only able to manipulate finite numbers, when computing the partial derivative during the gradient calculation, if the weights are too small they will eventually collapse to zero. Therefore, the gradient vanishes because the partial derivatives from which it is composed quickly go to zero.

and a hyperbolic tangent function which creates a vector for possible values that could be added to the cell state. Thirdly, the cell state (the upmost line) is updated by discarding the information that should be discarded and by adding the information decided in the previous step. Finally, the output gate decides which data to let flow outside by applying the down most sigmoid and the rightmost hyperbolic tangent.

A last advancement in this topic are Gated Recurrent Units (GRUs in short) [21], depicted in Figure 3.3. These are composed as an LSTM cell, with one main difference: while LSTMs have three gates, GRUs have only two, which are the input and the forget gate. Having them a gate less, they are considered far easier to modify and the time they take for training is usually much inferior than standard LSTMs.

3.3 Encoder-Decoder networks

Encoder-Decoder networks are, as the name suggests, neural networks constituted by two main components: an encoder and a decoder. These kinds of architectures are usually used for sequence-to-sequence (seq2seq in short) problems, *i.e.* when inputs and outputs have different dimensions and they are unaligned, that is the ordinality of input and output are different; typical applications of these models are therefore machine translation, image captioning, sentiment analysis and similar tasks.

The encoder of such a network is usually composed of one or more recurrent units, such as the RNNs and LSTMs cited in Section 3.2; these units accept an element from the input sequence and they push it through the encoder network, grasping information in the meanwhile. At the end of this network, the remaining output is just one vector: this is called *encoder vector*, also known as *hidden state* of the Encoder-Decoder network. The underlying idea is that this vector contains a representation of all the data which has been pushed through the network; in other words, it embeds the information from all the input elements. In mathematical terms, the hidden state is defined by the following:

$$h_t = f(\mathbf{W}_{hh} h_{t-1} + \mathbf{W}_{hx} x_t) \tag{3.5}$$

where t is the last step of the encoder, and $\mathbf{W}_{hh}$ and $\mathbf{W}_{hx}$ are the weight matrices for, respectively, the previous hidden state and the last input sequence.

This vector is then fed to the decoder network. As for the encoder, this is usually composed by one or more recurrent units; each of these receives a hidden state from the previous one and produces another hidden state. The final output is then the actual output, which is different concerning the specific domain the network is used. In mathematical terms, each hidden state is computed by the following:

$$h_t = f(\mathbf{W}_{hh} h_{t-1}) \tag{3.6}$$

The final output is then computed by:

$$y_t = \text{softmax}(\mathbf{W}_s h_t) \tag{3.7}$$

As outlined, encoder-decoder networks are often used to generate some output starting from input sequences; this fact makes them enter the category of generative models. These are statistical models which, informally, can generate new data starting from some given training. More formally, generative models are able to grasp the joint probability $P(X, Y)$ where X is the given data and Y are the labels for this information. There exist thousands of generative models; in the next sections, we will analyze the ones which will be needed to understand the founding works of this thesis.

3.3.1 Autoencoders and Variational Autoencoders

Autoencoders [22] are a particular case of Encoder-Decoder networks: as Figure 3.4 suggests, their architecture is almost the same and their encoders and decoders are designed in the same way. However, while in Encoder-Decoder networks the output can be possibly everything, in autoencoders it is, or it should be, equal to the input. The underlying idea of autoencoders is, therefore, to learn an efficient representation of the input by reducing its dimensionality: this representation corresponds to the hidden state and it has, indeed, a smaller size than the inputs. In order to enable this, the two networks are trained by minimizing the distance between the initial output x and the final output x', with loss functions such as the mean squared error. Thanks to their properties, autoencoders are typically used for dimensionality reduction techniques (such as principal component analysis, or PCA) and binary compression.

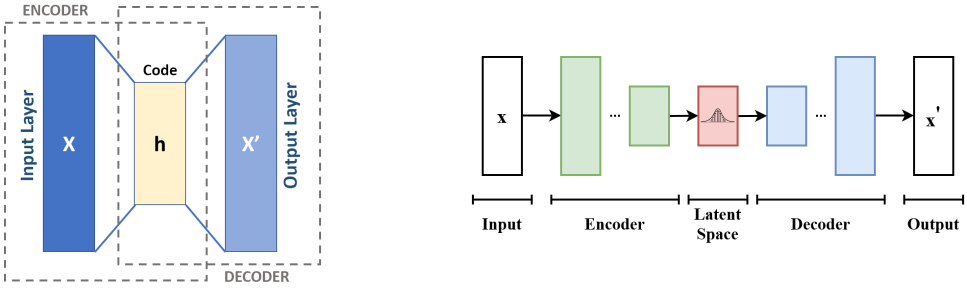

Figure 3.4: The standard architecture for Autoencoders (left) and Variational Autoencoders (right). Source: wikimedia.org.

An advancement of autoencoders are Variational Autoencoders (VAEs) [23]. In this family of models, the hidden vector is replaced with the parameters of a probability distribution in the latent space; therefore, the encoder and the decoder networks work the same way as simple autoencoders (cf. Figure 3.4), but the concept is different. Instead of having a fixed hidden representation, in fact, we have two parameters μ_x and σ_x: these are respectively the mean and the standard deviation for a Gaussian distribution of the input hitherto seen; the input to the decoder is therefore a sample $z \sim \mathcal{N}(\mu_x, \sigma_x)$ from the resulting distribution.

These two formulations might appear as almost equal, but the difference is notable. Being the hidden representation a probability distribution, VAEs can be (and often are) used as generative models: in fact, after having trained a VAE, the probability distribution can be sampled in order to obtain unseen data. This notion will become paramount in the subsequent sections and chapters since most of the works this thesis is based on use variations of VAEs to model their architectures.

3.3.2 Attention

A last concept has now to be explored before discussing Transformer, which is the starting model for many of the works in the trajectory prediction field including the foundational works of this thesis: this concept is the attention in machine learning.

Attention, first defined by [24], is a technique that aims to solve a particular problem in Encoder-Decoder networks: the fixed-size representation of the hidden state. Having a vector with a fixed dimension, in fact, might work for sequences under a certain threshold long, but it cannot be universal: there would be a sequence which is too long to be well repre-

sented by the compact hidden vector. The authors of the cited work, in particular, noticed that encoder-decoder networks were badly performing when translating long sequences of text from one language to another. The idea they had, then, was to develop a mechanism which would enable the model to allow all the hidden states from the encoder to contribute to the decoding phase. In order to do so, the mechanism uses three sets, namely queries, keys and values, to compute a context vector that is fed to the decoder at each step. The general procedure is to start by computing what the authors call the alignment scores, then compute the weights and finally calculate the context vector.

More specifically the alignment scores are computed by comparing each query vector q, namely the output from the previous output from the decoder s_{t-1}, with a database of keys; this corresponds to the following:

$$e_{q,k_i} = q \cdot k_i \tag{3.8}$$

From these scores, the weights are computed by applying a softmax function:

$$\alpha_{q,k_i} = \text{softmax}(e_{q,k_i}) \tag{3.9}$$

Finally, the attention is computed by summing the weighted value vectors:

$$\text{ATT}(q, \mathbf{K}, \mathbf{V}) = \sum_i \alpha_{q,k_i} v_{k_i} \tag{3.10}$$

The procedure just described was originally thought for machine translation or, in general, for tasks in the natural language processing domain. However, in the last years, attention mechanisms achieved great popularity in the computer vision field for their capability in understanding visual cues; an example of this is *Goal-SAR*, described in Section 3.4, and the other related works listed in Chapter 2. Moreover, many advancements have been done, as reported in the next section.

3.3.3 Transformer

One of the biggest projects which strongly use attention mechanisms is Transformer [25]; this work, indeed, claims that recurrent networks and convolutions are not needed when dealing with learning sequences: Attention is all you need, as the title suggests. In order to do so, the authors provide two additional types of attention, which are hereby described:

scaled-dot product attention and multi-head attention.

Scaled dot-product attention consists in the computation of the dot products of the query for all the keys and its division by $\sqrt{d_k}$, where d_k is the dimension of the keys. In mathematical terms, this is equivalent to:

$$\mathrm{ATT}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{3.11}$$

where $\mathbf{Q}$ is the matrix equivalent to the set of queries, $\mathbf{K}$ are the keys and $\mathbf{V}$ are the values.

On the other hand, Multi-head attention consists in the application of the attention mechanism h times. In order to do so, a linear projection of the queries, keys and values is performed at each time; these linear projections are learned and they change at each time. All of this is performed in parallel; every projection is then concatenated and projected again to get the final results. In mathematical terms, this corresponds to:

$$\mathrm{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{concat}(h_1, ..., h_h)\mathbf{W^O} \tag{3.12}$$

where

$$h_i = \mathrm{ATT}(\mathbf{Q}\mathbf{W}_i^{\mathbf{Q}}, \mathbf{K}\mathbf{W}_i^{\mathbf{K}}, \mathbf{V}\mathbf{W}_i^{\mathbf{V}}) \tag{3.13}$$

with $\mathbf{W}_i^{\mathbf{Q}}$, $\mathbf{W}_i^{\mathbf{K}}$, $\mathbf{W}_i^{\mathbf{V}}$ and $\mathbf{W^O}$ being the projections of respectively the queries, the keys, the values and the output.

A Transformer model has an Encoder-Decoder structure, which we described in Section 3.3; instead of using RNNs, LSTMs or GRUs, it uses stacked self-attention and fully connected layers as depicted in Figure 3.5 and here described.

The encoder consists of a stack of N identical layers, which in the paper are originally $N = 6$, each of them having two sublayers: a multi-head attention mechanism and a fully connected neural network. In addition to this, a residual connection and a layer normalization are applied to both sublayers. Similarly to the encoder, the decoder is composed by N stacked layers, and it employs multi-head attention followed by residual connections and normalizations. Furthermore, a third sub-layer is a multi-head attention over the output of the encoder. Moreover, the self-attention sublayer is modified in order for it to not access

subsequent positions in the sequence.
The three remaining components are the feed-forward networks, the embeddings with the corresponding softmax and the positional encoding. With regards to the former, these are applied both in the encoder and decoder after the two attention sublayers; both of them consist in two linear transformations with a ReLU in the middle. Similarly, the embeddings are used to convert the input and output tokens into a fixed size, and the softmax function is used to convert the output to predicted probabilities of which will be the next token. Finally, in order for the model to grasp the ordinality cues, the positional encodings are added to the input embeddings; in this specific case, these are sine and cosine functions.

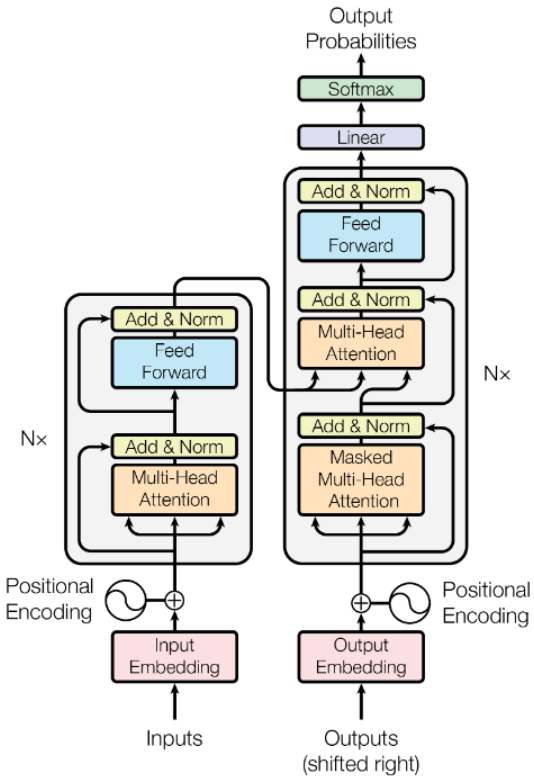

Figure 3.5: The architecture of a Transformer model. Source: https://arxiv.org/abs/1706.03762.

3.4 Goal-SAR

Goal-SAR, short name for *Goal-driven Self-Attentive Recurrent Networks for Trajectory Prediction* [5], belongs to the category of works in the trajectory prediction domain which try to exploit both scene information and final goals to generate plausible trajectories; it is de-

veloped by the VIMP group at University of Padova and is currently improving. The core idea of this work is that predicting probable destinations should give additional information to the model, allowing it to generate more realistic paths; this is done in parallel with an attention-based recurrent temporal backbone, which naturally models the progression of a trajectory, as explained in the next section.

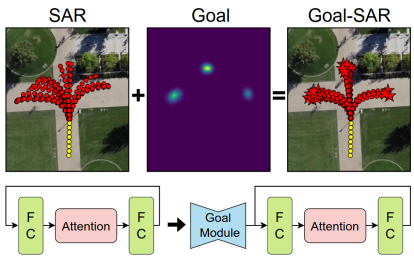

Figure 3.6: Visualization of Goal-SAR main idea. Source: https://arxiv.org/abs/2204.11561.

As the reader can see from Figure 3.6, and as the work's name suggests, this model makes important use of the concepts explained in Sections 3.2 and 3.3.

3.4.1 Model architecture

The model architecture is composed by two modules: an attention-based temporal recurrent backbone and a goal estimation module; these work independently for most of the time, but their results are put together at the end of a training iteration to optimize a common loss. The architecture is summarized in Figure 3.7: the upmost part is the temporal backbone, while the down most is the goal module.

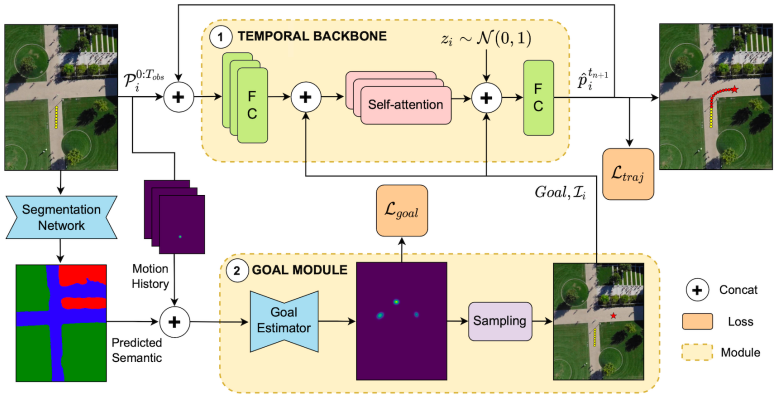

Figure 3.7: The Goal-SAR model architecture. Source: https://arxiv.org/abs/2204.11561.

The main contributions this model brings are therefore two: the recurrent backbone, which is alone able to outperform several other competing models, and the goal module which enables the model to produce even better results. The way each component works is described in the next paragraphs.

Recurrent backbone

The recurrent backbone has a VAE-like structure, with an encoder which processes temporal dependencies and a decoder that actually outputs the predicted trajectories. Temporal dependencies are the only information which is taken into account by this module; this latter processes the two-dimensional sequences by encoding them into a feature space. The authors report this procedure as follows:

$$e_i^t = \phi(p_i^t, \mathbf{W}_e) \tag{3.14}$$

where p_i^t are the (x, y) coordinates of agent i at timestep t, ϕ is a linear embedding function with a non-linearity and $\mathbf{W}_e$ are the embedding weights.

With regard to the encoder, this consists in the encoding part of a Transformer model, already described in Section 3.3.3. The use of such a model allows to subsequently predict the future positions starting from a variable-size input sequence. From a technical perspective, the temporal dependencies are considered by applying a linear projection of the embeddings e_i^t to obtain a triplet of vectors q_i^t, k_i^t, v_i^t, respectively query, key and value. These are used to compute the output of the encoder, which is defined as follows:

$$\mathrm{ATT}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) = \mathrm{softmax}\left(\frac{\mathbf{Q}_i \mathbf{K}_i^t}{\sqrt{d_k}}\right)\mathbf{V}_i \tag{3.15}$$

where d_k is the normalization factor. Reportedly, this operation is performed N_{head} (experimentally set to $N_{head} = 8$) on different projections. The final output is then a vector of embedded positions $(h_i^0, ..., h_i^t)$ within which the temporal dependencies are represented. The reader may notice that this equation is very close to the ones reported in Sections 3.3.2 and 3.3.3.

The last vector is then given to the decoder, which is defined as a function ψ composed by a linear layer and a non-linearity (ReLU) with $\mathbf{W}_d$ weights. This firstly concatenates a

Gaussian random vector $\mathbf{z}_i \sim \mathcal{N}(0, \mathbf{I})$ in order to increase the diversity of the generated sequences, and then it applies the decoding to generate the predicted positions; this is summarized by the following:

$$p^{t_{n+1}} = \psi(\text{concat}(\mathbf{h}_i^{t_n}, \mathbf{z}_i); \mathbf{W}_d) \tag{3.16}$$

As the reader may grasp from the formula above, the sequences are not generated as-is but this process is performed iteratively: the estimated locations are recursively concatenated to the previous input sequence.

Goal Module

As previously outlined, the goal-estimation module aims to produce a probability map of the final destination of an agent. In order to work, the goal module needs two different types of information: the semantics from the scene and the observed positions. The former is retrieved by the RGB images of the scenes: these are fed to the semantic segmentation network from [6], which classify each pixel as one class from the set $C = \{$*pavement*, *terrain*, *structure*, *tree*, *road*, *not defined*$\}$; the output is a tensor $\mathbf{S} \in \mathbb{R}^{W \times H \times C}$, with W, H being the dimensions of the images and C the semantic classes. The latter is a tensor of size $W \times H \times N_{obs}$, with N_{obs} being the number of observed steps in the history path, and it consists in a two-dimensional probability distribution $\mathcal{N}(p_i^t, \sigma_s^2 \mathbf{I})$ which denotes the observed positions on a two-dimensional map. The authors define this projection as $\mathcal{M}(p_i^t)$.

The two tensors are then concatenated and fed to an ad-hoc U-Net network which outputs a two-dimensional heat map, of which a sample represents the probabilities of an agent to finish its path in a certain point given its past motion history: this sample is taken from the aforementioned probability map and is injected into the temporal backbone.
As a final note, the U-Net consists in L blocks that reduce the dimensions via convolutions and max-pooling operations, and passes each output in the range $[1, L]$ to the decoder via skip-connections. Moreover, the output from the final block is doubled in resolution by bilinear up-sampling, convolutions and non-linearity. This latter and the former are fused by a convolutional layer and a sigmoid function, returning the final probability distribution of the final position.

3.4.2 Training objective

Being the architecture composed by two independent modules, two different loss functions will need to be optimized: one for the goal module and one for the recurrent backbone. The training objective for the former consists in optimizing a Binary Cross-Entropy loss between the predicted probability map and the ground truth, as defined in Equation 3.17; these are obtained as two two-dimensional Gaussian distributions $\mathcal{N}(p_i^{T_{seq}}, \sigma_s^2 \mathbf{I}_2)$ centered at the ground truth final destination.

$$L_{goal} = \frac{1}{N_p} \sum_{i=1}^{N_p} \text{BCE} \left(\mathcal{M} \left(p_i^{T_{\text{seq}}} \right), \hat{\mathcal{M}} \left(p_i^{T_{\text{seq}}} \right) \right) \tag{3.17}$$

The recurrent backbone, on the other hand, is trained by optimizing the loss function listed in Equation 3.18. This consists in the Mean Squared Error between the predicted and the real positions for each point in the future points' set.

$$L_{traj} = \frac{1}{N_p \cdot T_{\text{pred}}} \sum_{i=1}^{N_p} \sum_{t=T_{\text{obs}}+1}^{T_{\text{seq}}} \left\| p_i^t - \hat{p}_i^t \right\|_2^2 \tag{3.18}$$

These two functions are then put together for the actual optimization of the network; the final loss function is therefore defined as follows:

$$L = L_{goal} + \lambda L_{traj} \tag{3.19}$$

where λ is a hyperparameter that controls each network's contribution; the authors of the paper found it beneficial (experimentally) to put it equal to $\lambda = 10^{-6}$.

3.5 Diffusion models

Diffusion models, first appeared in [2], are probabilistic and generative models inspired by non-equilibrium thermodynamics. The main idea is to start from a known data distribution and slowly destroy it by applying some noise; this noise usually consists in a sample from a normal distribution with zero mean and unit variance[2]. After the data has been destroyed,

[2]Also known as Gaussian distribution, it is the probability distribution given by $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$ where $\mu = 0$ and $\sigma = 1$ (or $\sigma = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix, when the distribution is multi-dimensional). Usually wrote $\mathcal{N}(0, 1)$ in the case of a one-dimensional distribution or $\mathcal{N}(0, \mathbf{I})$ in the multi-dimensional case.

we then learn a reverse diffusion process which re-establishes the original data; this can be achieved in different ways, but the common procedure is to approximate the parameters of the posterior distribution by optimizing a loss function using a deep neural network. This entire process can also be seen as a Markov chain[3], which is traveled in one direction during the noising phase and in the opposite direction during the denoising phase, as shown in Figure 3.8.

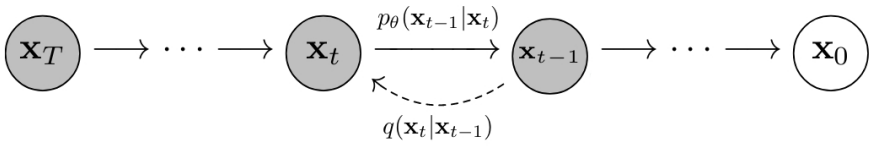

Figure 3.8: The Markov chain visualization of the diffusion processes. Source: https://arxiv.org/abs/2006.11239.

A detailed description of the two processes of diffusion (forward and reverse) and the typical loss function now follows.

Forward process

The forward process, as mentioned earlier, consists in a Markov chain where we gradually add noise (namely, samples from a Gaussian distribution) to the original data; we do this according to a variance schedule $\beta_1, ..., \beta_T$. The number T is the number of diffusion steps, *i.e.* how many samples we take from the Gaussian distribution; with variance schedule we mean a function that controls the intensity of the noise (*i.e.* the weight for the sample at diffusion step $t \in [1, T]$). Both of these are experimentally obtained, and they can possibly differ according to the application domain as extensively discussed in Chapter 5.

This process can therefore be described by the following conditioned probabilities:

$$q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right) = \prod_{t=1}^{T} q\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}\right) \tag{3.20}$$

where

$$q\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}\right) = \mathcal{N}\left(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta t \mathbf{I}\right) \tag{3.21}$$

[3] In general, a stochastic model which describes possible events; each event's probability depends only on the state of the previous event.

This means that the final, noised data is equal to the product of the sequence of $q(\mathbf{x}_t \mid \mathbf{x}_{t-1})$, which in turn are the result of a noising step. Note that $\mathbf{x}_1, \ldots, \mathbf{x}_T$ are latent variables with the same dimensionality as the original data $\mathbf{x}_0$.

This process can be done iteratively; computationally speaking, this equals a complexity of $\mathcal{O}(T)$. However, there exists a closed form of the equation that enables us to sample a general $\mathbf{x}_t$, and therefore the final $\mathbf{x}_T$, in just one step; note in fact that

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}\left(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}\right) \tag{3.22}$$

and then

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon \tag{3.23}$$

where

$$\begin{aligned} &\bar{\alpha}_t = \textstyle\prod_{s=0}^{T} \alpha_s, \\ &\alpha_t = 1 - \beta_t, \\ &\varepsilon \sim \mathcal{N}(0, \mathbf{I}) \end{aligned} \tag{3.24}$$

This way, once a variance schedule is decided, we can easily sample the final data distribution by applying the noise in just one step. The entire derivation for this closed formulation is described in [3].

Reverse process

As previously mentioned, the reverse process consists of a parametrized Markov chain with learned transitions. In this phase, we gradually remove noise from the corrupted data distribution in order to return to the original distribution; while doing so, we gradually learn the parameters that will be needed to generate new data by optimizing the loss function explained in the next paragraph. This process is described by the following:

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) \tag{3.25}$$

where

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) = \mathcal{N}\left(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t)\right) \tag{3.26}$$

There are a few aspects of these formulas that need to be clarified: first of all, note that θ consists in the vector of parameters that we try to learn, and then p_θ is the posterior distribution we should achieve after the entire process. Consequently, there cannot be a closed form of these formulas, because the process must be iterative (as better explained in the next paragraph). Moreover, note that we do not try to compute the parameters for p_θ in a direct fashion: instead, we learn the parameters for μ and Σ, which are respectively the mean and the variance of the Gaussian distribution.

After this process, we can use the resulting parametrized distribution to generate new data: not surprisingly, diffusion models are generative models. As we will see in Section 3.5.1 and in Chapter 5, the main idea is to feed the distribution unseen data (*i.e.* a random Gaussian distribution), and this should enable us to obtain different data from the original data distribution.

Loss function

In order to obtain the parameters of the probability distribution p we need to train a model by maximizing the model log-likelihood (or, respectively, minimize the negative log-likelihood); in mathematical terms, we try to minimize

$$\mathbb{E}\left[-\log p_\theta(\mathbf{x}_0)\right] \tag{3.27}$$

In order to achieve this, we can optimize the variational bound provided by Jensen's inequality[4]; this has been already proved in [2]. By this property, we get the following:

$$\begin{aligned} \mathbb{E}\left[-\log p_\theta(\mathbf{x}_0)\right] &\leq \mathbb{E}_q\left[-\log \tfrac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\right] \\ &= \mathbb{E}_q\left[-\log p(\mathbf{x}_T) - \textstyle\sum_{t\geq 1} \log \tfrac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_t|\mathbf{x}_{t-1})}\right] \end{aligned} \tag{3.28}$$

The resulting expectation coincides with the theoretical loss function. However, this form is intractable from a computational perspective; therefore, in literature we always find the

[4] In short: for every function g we have that $\mathbb{E}[g(X)] \geq g(\mathbb{E}[X])$.

version which makes use of Kullback-Leibler (KL) divergence[5] to compute the factors in a tractable way. This formula is here reported; for the complete derivation, the reader might refer to [2], Appendix B:

$$\begin{aligned} L_{vlb} = \mathbb{E}_q[\underbrace{D_{\mathrm{KL}}\left(q\left(\mathbf{x}_T \mid \mathbf{x}_0\right) \| p\left(\mathbf{x}_T\right)\right)}_{L_T} \\ + \sum_{t>1} \underbrace{D_{\mathrm{KL}}\left(q\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0\right) \| p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)\right)}_{L_{t-1}} \\ \underbrace{-\log p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)}_{L_0}] \end{aligned} \tag{3.29}$$

As a last remark, note that this loss function is only a general formula. Indeed, there exist different and specific formulations depending on the application domain it needs to be used for; one of them is described in Section 3.5.1, but its customization is also part of this work itself as reported in Chapters 5 and 6.

3.5.1 Denoising Diffusion Probabilistic Models

Denoising Diffusion Probabilistic Models [3] is probably the most famous work when we talk about the application of diffusion models to practical problems. While the authors of the original paper [2] invented and studied this concept for relatively simple data such as a two-dimensional swiss roll, handwritten digits (MNIST, [26]) and small natural images (CIFAR-10, [27]), the authors of *DDPM* have been able to use it for high-quality image synthesis. In order to do so, they further developed diffusion models and adapted the concept to this specific task.

Even though the authors apply the concept of diffusion to image synthesis, this work is strongly related to the founding paper of this work, which is reported in Section 3.6: in fact, it is, in turn, the point from which the authors of *MID* started to develop their work. In the next paragraphs, an overview of the most important implementations and improvements is provided, since it will be needed to discuss the improved version of *DDPM*, and therefore this thesis' effort, in Chapter 5.

[5] A type of statistical distance which measures how one probability distribution differs from a second one. Given two distributions P and Q, it is defined as $D_{KL}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) log\left(\frac{P(x)}{Q(x)}\right)$ when operating in a discrete set or $D_{KL}(P \parallel Q) = \int_{-\infty}^{\infty} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$ for a continuous probability distribution.

Forward process

In their specific implementation, the authors decided to ignore that the variances β_t for the forward process are learnable; conversely, they decided to fix them to constants, which are computed just once before the start of the actual forward process. To be more specific, they opted for a linear schedule where the variances increase from a minimum of $\beta_1 = 10^{-4}$ to a maximum of $\beta_T = 0.02$. As discussed in the next paragraphs, this decision came from the fact that their experimental results showed that learning both the mean and the variance for the posterior distribution did not improve the results; instead, their model only learns the means.

Moreover, they chose this specific range of values because considered small relatively to their scaled data; this choice ensures that the forward and the reverse process have a similar functional form. As a last point, we report that both their forward and reverse process consist in $T = 1000$ diffusion steps.

Reverse process

As mentioned in the forward process paragraph, a first implementation has been to fix the variances. For the reverse process, therefore, the authors do not consider the variances to be learnable; instead, they fix them to

$$\Sigma(\mathbf{x}_t, t) = \sigma_t^2 \mathbf{I} \tag{3.30}$$

where

$$\begin{aligned} \sigma_t^2 &= \tilde{\beta}_t \\ \tilde{\beta}_t &= \frac{1 - \tilde{\alpha}_{t-1}}{1 - \tilde{\alpha}_t} \beta_t \end{aligned} \tag{3.31}$$

Moreover, they decided on a reparametrization of the mean μ_θ in order to adapt it to the data available at run-time and to have an easier-to-implement and easier-to-optimize loss function. The entire derivation can be seen in [3]; the final result is the following:

$$\begin{aligned} \mu_\theta(\mathbf{x}_t, t) &= \tilde{\mu}_t \left(\mathbf{x}_t, \frac{1}{\sqrt{\bar{\alpha}_t}} \left(\mathbf{x}_t - \sqrt{1 - \bar{\alpha}_t} \epsilon_\theta(\mathbf{x}_t) \right) \right) \\ &= \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_\theta(\mathbf{x}_t, t) \right) \end{aligned} \tag{3.32}$$

A crucial point to note in the previous formulation is that now the parametrized mean depends on a parametrized ϵ_θ; this term consists in an approximator function for ϵ, which in turn is the Gaussian noise that needs to be subtracted from the noisy distribution in order to get the original data distribution. In practice, this fact means that the model for *DDPM* does not try to recreate directly the original distribution, but instead it needs to learn the noise to remove from the corrupted data in order to get the desired distribution.
As a last note, we report that the approximator function in this implementation consists in a U-Net, which is trained by optimizing the loss function reported in the following paragraph.

Loss function

As stated in Section 3.5, the general loss function obtained by computing the KL-divergence terms is tractable: it can be used to optimize a model since it is differentiable with respect to θ, and thus it can be solved by applying a gradient descent algorithm. However, the authors of *DDPM* found it beneficial in sample quality terms to optimize a way simpler version of the variational bound. This has been called L_{simple}, and it is defined by the following:

$$L_{\text{simple}}(\theta) = \mathbb{E}_{t,\mathbf{x}_0,\epsilon}\left[\left\|\epsilon - \epsilon_\theta\left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t\right)\right\|^2\right] \tag{3.33}$$

where ϵ is pure Gaussian noise (sampled from a zero mean and unit variance Gaussian distribution) and $t \in [1, T]$ is the diffusion step. In practice, with this loss function, we are trying to reduce the mean squared error between the pure Gaussian noise and the predicted noise by optimizing the parameters of the latter.

Algorithms and example

Now that the custom implementations for the forward and reverse processes have been discussed, we can present the algorithms for training and sampling.

With regards to the training phase, listed in Algorithm 3.1, note that it is assumed that the data have already been through the forward process, thus it already consists in the corrupted distribution. In order to learn the parameters for the final distribution, the gradient descent algorithm takes a step in the direction which minimizes the mean squared error between the standard Gaussian noise and the parametrized noise.

Algorithm 3.1 Training phase. Source: https://arxiv.org/abs/2006.11239.

while not converged
  $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
  $t \sim \text{Uniform}(\{1, ..., T\})$
  $\epsilon \sim \mathcal{N}(0, \mathbf{I})$
  take gradient descent step on $\nabla_\theta \left\| \epsilon - \epsilon_\theta \left(\sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon, t \right) \right\|^2$

Similarly, for the sampling algorithm it is assumed that the model has been trained with the algorithm above. During this phase, we feed standard Gaussian noise to the for cycle and, for each diffusion step, we perform a backward step on the corresponding Markov chain; this way, when we finish the denoising process, $\mathbf{x}_0$ will be the resulting data.

Algorithm 3.2 Sampling phase. Source: https://arxiv.org/abs/2006.11239.

$\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$
for $t = T, ..., 1$
  if $t > 1$
    $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$
  else
    $\mathbf{z} = 0$
  $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \epsilon_\theta \left(\mathbf{x}_t, t \right) \right) + \sigma_t \mathbf{z}$
return $\mathbf{x}_0$

To conclude this section, a toy example of the forward and reverse diffusion processes is provided in Figure 3.9. The first row corresponds to the forward process applied to an image; the model is then trained on many batches of data using the listed algorithm. Finally, the sampling algorithm (second row in the figure) is used to generate new data.

Figure 3.9: An example of forward diffusion and sampling. Source: https://arxiv.org/abs/2006.11239.

3.6 Motion Indeterminacy Diffusion

As already mentioned *MID*, short for *Stochastic Trajectory Prediction via Motion Indeterminacy Diffusion* [1] constitutes the foundational work for this thesis: its theoretical results, as well as the code, are used as a basis for this project. In the cited work, the authors apply the diffusion theory coming from *DDPM* to the trajectory forecasting domain; as previously outlined, their idea is to simulate the intrinsic human indeterminacy with Gaussian noise, and consequently removing this noise should enable the generation of realistic trajectories. This concept is well visualized in Figure 3.10: at the end of the forward process, the trajectory distribution is essentially just Gaussian noise, while at the end of the reverse process the trajectory is a small and plausible distribution.

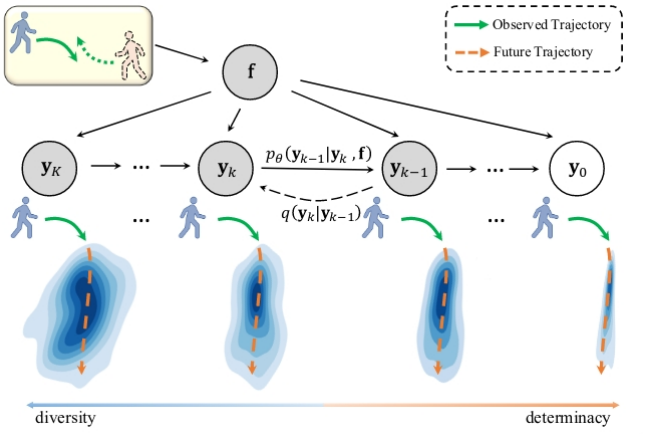

Figure 3.10: Visualization of diffusion for trajectory prediction in MID: diversity vs determinacy. Source: https://arxiv.org/abs/2203.13777.

In this section we will first analyze the general architecture of the model and its pipeline; after this, we will describe the specific implementations of *DDPM* concepts to the trajectory framework.

3.6.1 Model architecture

As with most of the models in the literature, *MID* is composed of several parts, which cooperate for the final goal. From a mere code organization perspective, the project is composed by two phases: a first data pre-processing phase, where the training and testing data is pre-

pared for the task and auxiliary information (such as velocity and acceleration) is computed, and proper training and inference phase.

Most importantly, the general structure of the model is a Transformer-based network which acts as a variational autoencoder, as illustrated in Figure 3.11. In particular, the encoder and decoder parts are constituted as hereby described.

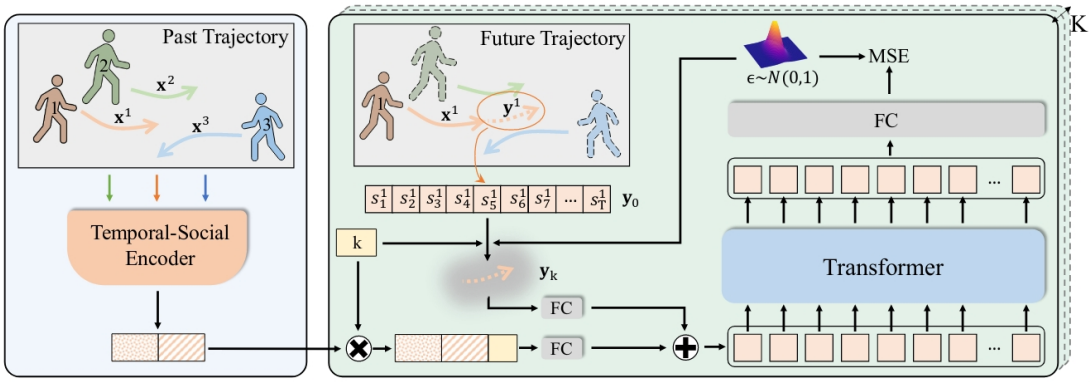

Figure 3.11: Architecture of MID. Source: https://arxiv.org/abs/2203.13777.

Encoder

The encoder is a temporal-social encoder network; this component maps the history path and the social interaction information into an embedding $\mathbf{f}$, which is then fed to the decoder. Note that the social interaction clues are not given, but it is supposed that the encoder is able to grasp this information; in fact, this component is taken from the already cited *Trajectron++*, which has this exact capability. However, as the authors report, *MID* is an encoder-agnostic framework; that is, the network is not strictly designed to model social interactions and any other encoder network can be equipped.

Decoder

The decoder takes the data $\mathbf{y}_k$ (corrupted k times), the state embedding and the time embedding and generates two outputs; these are the means μ_x, μ_y for the multivariate Gaussian distribution which models the predicted noise. In other words, this network models the $\epsilon_{(\theta,\psi)}$ from Equation 3.37, which is conceptually the same as Equation 3.33.
This decoder consists in a Transformer-based architecture, and as already mentioned it models the Gaussian transitions in the associated Markov chain. Formally, its inputs consist in

the ground truth $\mathbf{y}_0$, the noise variable which is a sample from a standard Gaussian distribution, the feature $\mathbf{f}$ from the encoder, and a time embedding. At each step k, it adds noise to the trajectory to get the corrupted data $\mathbf{y}_k$, it computes the time embedding and it concatenates this latter to the feature. Then, a fully connected layer is used to upsample $\mathbf{y}_k$ and $\mathbf{f}$ and fuse them, together with the positional embedding, into a single tensor. Finally, this tensor is fed to the Transformer to learn the spatial-temporal information.

From a specific implementation point of view, this decoder is composed by three self-attention layers. A final fully connected layer is used to downsample the output from the Transformer and output the aforementioned means for predicted noise.

3.6.2 Diffusion

The two diffusion processes of *MID* looks alike the ones of *DDPM*, with three main differences: the first two, which also formally change the processes themselves, are that we are dealing with trajectories instead of images and we have the additional variable $\mathbf{f}$; the third, which is only conceptual, is that the noise we are adding should be considered as indeterminacy rather than only noise.

With this in mind, the forward process is regulated by the following closed form:

$$q\left(\mathbf{y}_k \mid \mathbf{y}_0\right) = \mathcal{N}\left(\mathbf{y}_k; \sqrt{\bar{\alpha}_k}\mathbf{y}_0, \left(1-\bar{\alpha}_k\right)\mathbf{I}\right) \tag{3.34}$$

where $\mathbf{y}_k$ and $\mathbf{y}_0$ are respectively the corrupted and the ground truth future trajectory, and

$$\begin{aligned} \bar{\alpha}_k &= \prod_{s=1}^{k} \alpha_s \\ \alpha_k &= 1 - \beta_k \end{aligned} \tag{3.35}$$

with $\beta_1, ..., \beta_k$ being the fixed variance schedules that constitute the weights for the applied noise.

On the other hand, the reverse diffusion process, and therefore the trajectory generation,

is regulated by the following:

$$
\begin{aligned}
p_\theta\left(\mathbf{y}_{0:K} \mid \mathbf{f}\right) &:= p\left(\mathbf{y}_K\right) \prod_{k=1}^{K} p_\theta\left(\mathbf{y}_{k-1} \mid \mathbf{y}_k, \mathbf{f}\right) \\
p_\theta\left(\mathbf{y}_{k-1} \mid \mathbf{y}_k, \mathbf{f}\right) &:= \mathcal{N}\left(\mathbf{y}_{k-1}; \mu_\theta\left(\mathbf{y}_k, k, \mathbf{f}\right); \mathbf{\Sigma}_\theta\left(\mathbf{y}_k, k\right)\right)
\end{aligned}
\tag{3.36}
$$

where $\mathbf{f}$ is the feature state embedding learned by the temporal-social encoder $\mathcal{F}_\psi$ and the variance $\mathbf{\Sigma}_\theta\left(\mathbf{y}_k, k\right)$ is set to $\beta_k \mathbf{I}$ due to the fixed variance schedule.

3.6.3 Training objective

As should be clear from the previous paragraphs, *MID* generates the trajectory by removing the learned noise from a standard Gaussian distribution with zero mean and unit variance. In order to build the final noise distribution its parameters are learned, along with the parameters of the temporal-social encoder, by optimizing an ad-hoc version of Equation 3.33; this function is defined as follows:

$$
L(\theta, \psi) = \mathbb{E}_{\epsilon, \mathbf{y}_0, k} \left\| \epsilon - \epsilon_{(\theta,\psi)}\left(\mathbf{y}_k, k, \mathbf{x}\right) \right\| \tag{3.37}
$$

where ϵ is a sample from a standard Gaussian distribution, $\mathbf{y}_k = \sqrt{\bar{\alpha}_k}\mathbf{y}_0 + \sqrt{1-\bar{\alpha}_k}\epsilon$, θ are the parameters of the final noise distribution and ψ are the parameters from the temporal-social encoder.

The training is performed for each diffusion step k; the procedure can be summarized by Algorithm 3.3.

Algorithm 3.3 Training phase of MID. Source: https://arxiv.org/pdf/2203.13777.

1: **while** not converged
2: $(\mathbf{x}, \mathbf{y}) \sim q_{data}$ where $\mathbf{x}$ and $\mathbf{y}$ are respectively observed and future trajectory
3: $\mathbf{y}_0 = \mathbf{y}$
4: $k \sim \text{Uniform}(\{1, ..., K\})$
5: $\epsilon \sim \mathcal{N}(0, \mathbf{I})$
6: take gradient descent step on $\nabla_{(\theta,\psi)} \left\| \epsilon - \epsilon_{(\theta,\psi)}\left(\sqrt{\bar{\alpha}_k}\mathbf{y}_0 + \sqrt{1-\bar{\alpha}_k}\epsilon, k, \mathbf{x}\right) \right\|^2$

3.6.4 Inference

After the training phase, trajectories can be generated by removing the noise regulated by θ from pure Gaussian noise, as previously outlined. This procedure, summarized by Algorithm 3.4, consists in starting from a sample from a Gaussian $\mathbf{y}_k \sim \mathcal{N}(0, \mathbf{I})$ and iteratively removing the noise using the following:

$$\mathbf{y}_{k-1} = \frac{1}{\sqrt{\alpha_k}} \left(\mathbf{y}_k - \frac{\beta_k}{\sqrt{1-\bar{\alpha}_k}} \epsilon_\theta \left(\mathbf{y}_k, k, \mathbf{f} \right) \right) + \sqrt{\beta_k} \mathbf{z} \tag{3.38}$$

where $\mathbf{z}$ is a random variable in a standard Gaussian distribution and ϵ_θ is the learned noise coming from the network; this includes the previous diffusion step $\mathbf{y}_k$ and the feature state embedding $\mathbf{f}$.

Algorithm 3.4 Sampling a trajectory in MID. Source: https://arxiv.org/pdf/2203.13777.

for $k = K, ..., 1$
  if k > 1
    $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$
  else
    $\mathbf{z} = 0$
  $\mathbf{y}_{k-1} = \frac{1}{\sqrt{\alpha_k}} \left(\mathbf{y}_k - \frac{\beta_k}{\sqrt{1-\bar{\alpha}_k}} \epsilon_{(\theta,\psi)} \left(\mathbf{y}_k, k, \mathbf{x} \right) \right) + \sqrt{\beta_k} \mathbf{z}$
$\mathbf{y} = \mathbf{y}_0$

4

Datasets

There exist several datasets for the human trajectory forecasting task; however, most of the literature on the topic uses three of them: ETH/UCY, SDD, and inD. The average dataset for trajectory prediction research is composed by one or more scenes, *i.e.* aerial views of a particular and relatively small geographical area (for instance, a square); in each scene there can be one or more pedestrians, which are walking. The pedestrians are recorded in a sequence of frames, and each pedestrian follows one or more trajectories; the former case happens when the analyzed pedestrian remains in the area for all of the recording time, while the latter occurs when a pedestrian exits the scene and then re-enters the field of view.

The way data is recorded and annotated differs from dataset to dataset; for instance, the pedestrian's path can be recorded as the real distance (*i.e.* meters, centimeters) or as the relative distance in the scenes (*i.e.* pixels).
Since this work is based on [1], we use the same datasets; in fact, being the primary goal of this research to assess whether an advancement in the model can or can not improve the overall results, it would be inconsistent to use different datasets. For this reason, the used and analyzed datasets are ETH/UCY and SDD.

4.1 ETH/UCY

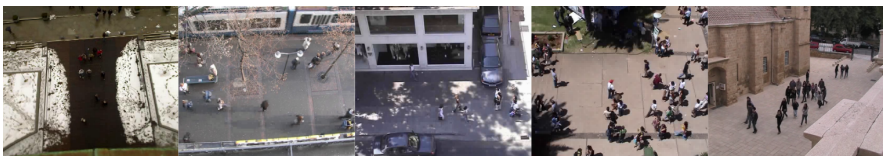

Figure 4.1: The five scenes from ETH and UCY datasets. From left to right: eth, hotel, zara1, univ, zara2. Source: https://arxiv.org/pdf/2010.00890.pdf.

ETH and UCY, first appeared respectively in [15] and [28], are the short names for two different datasets which are, however, often used together: ETH stands for ETH Zurich University, while UCY stands for the University of Cyprus.

These two datasets are composed by five different scenes: `eth` and `hotel` from ETH, `univ`, `zara1` and `zara2` from UCY. Each scene contains a variable number of frames, and each frame corresponds to $0.4s$ (thus the position of a pedestrian is sampled at 2.5 frames per second). The total amount of pedestrian trajectories is 1536.

In [1], and therefore in this work, the data is not adopted *as-is*, that is the raw videos are not used. Instead, only the annotations are used; these consist in text files composed by sequences of tuples of the type *(frame_id, track_id, x, y)*, where *frame_id* represents the frame of the video, *track_id* represents the trajectory of a pedestrian and *x, y* are the unnormalized coordinates in the time. The third dimension, *i.e.* the time, is automatically inferred since we already know that each point is recorded exactly $0.4s$ after the precedent one; the scene is also automatically inferred since each one of them is stored in a different file.

This raw data is then pre-processed by the program in order to create the batch data. During this phase, which is shared with the SDD pre-processing as discussed in the next section, the scene is created and additional information is added. In particular, the velocity and acceleration of a pedestrian are computed by differentiating the space dimension with regards to the time for respectively one and two times. Moreover, data augmentation [29] is performed in order to have more data to train and test the model; in particular, a partial rotation of the scenes is added to the final dataset.

Finally, the spatial coordinates are normalized by multiplying by 0.6 and subtracting the (x, y) position mean; this way, each pedestrian starting point is from the center of the scene.

4.2 SDD

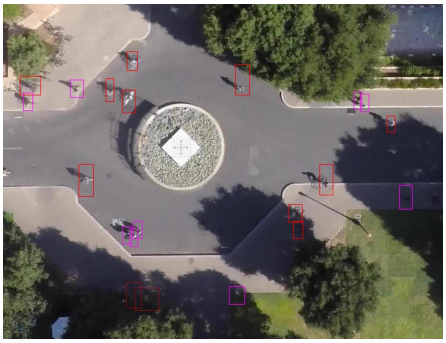

Figure 4.2: An example scene from SDD dataset. Source: https://academictorrents.com/details/01f95ea32e160e6c251ea55a87bd5a24b23cb03d.

SDD, firstly cited in [30], is the short name for Stanford Drone Dataset; the original version contains recordings of different types of moving nodes, namely pedestrians, bikers, cars, and other vehicles, but for this work we focus only on pedestrians. This dataset is composed by 47 scenes, divided into 8 macro-scenes; these scenes are listed in Table 4.1. As for ETH/UCY, each scene contains a variable number of frames, and each frame corresponds to $0.4s$.

Macro-scene	Number of scenes
bookstore	4
coupa	3
deathCircle	5
gates	9
hyang	10
little	4
nexus	10
quad	4

Table 4.1: Number of class labels for each image.

Like ETH/UCY, in [1] the data is not adopted as-is, but just the annotations are used after going through a pre-processing phase. The annotation files are composed of sequences of tuples of the type *(frame, trackId, x, y, sceneId, metaId)*, where *frame* represents the frame of the video, *trackId* represent the trajectory of a given node (since a single node can have more than one trajectory as discussed at the beginning of this chapter), *x, y* are the unnormalized coordinates in the time, *sceneId* represents the scene and *metaId* represents the pedestrian itself (since there can be more than one pedestrian in the same scene at the same moment).

As for ETH/UCY, the time dimension is automatically inferred. This raw data is also pre-processed by the program in order to create the batch data, though the procedure is slightly different; in this phase, the scene is created and velocity and acceleration information are added to the mere positional data. Data augmentation is also performed in the same way as for ETH/UCY. The difference with the previous dataset pre-processing consists in how the spatial coordinates are normalized: in this case, the coordinates are divided by 50 as [11] suggests.

5 Methods

As stated in the introduction to this work, the general methodology we followed has been iterative and incremental: being the *IDDPM* paper organized as improvements that can be applied to the original diffusion models, it has been easy to arrange the effort. This thesis' work can therefore be summed up in two different phases: in the former, which took approximately 70% of the endeavor, we tried to implement the improvements suggested in *IDDPM* into *MID*; in the latter, we tried to apply the findings of *Goal-driven Self-Attentive Recurrent Networks for Trajectory Prediction* to the model.

This chapter's structure follows the project timeline: in the first section we will analyze the existing problems we have found within the model; in the second we will analyze the incremental implementations from *IDDPM*; in the third, we will expose the breaking changing of the model with the implementation of the goal module. Finally, in the fourth, we analyze the long-term prediction scenario, which has been a side task and did not take a considerable amount of resources.

All the implementations have been made starting from the original *MID* code, published on Github by Gu Tianpei, at the following address: `https://github.com/Gutianpei/MID`. This project is entirely written in Python, and the neural network implementations are made using the PyTorch framework [31]. In order to maintain backward compatibility with the original code, every addition has been modularized and can be activated or deac-

tivated by just changing the respective flags in the configuration file. The final code, with all the implementations described in this work, can be accessed at the following address: `https://github.com/enricobu96/myMID`. For completeness, all the references we will talk about in this chapter, namely implemented classes, methods and functions, are symbolized by the `typographic font`; the file names are reported in square brackets, beside the corresponding reference.

5.1 Existing problems

Until now we talked about the positive aspects of *MID*; however, as well as every other work, it also has its downsides. Most of the problems lie in the scarce reproducibility of the experiments portrayed by the authors: the hyperparameters they report in their paper are, in fact, not likely to be the best ones. Instead, the parameters they provided on the project repository are most probably the ones which bring the best result. The most important different hyperparameter is the encoder dimension: in the paper the authors reported it to be 512, but the repository and the experiments showed that 256 was the optimal one. In addition to this, the authors did not provide the pre-trained models[1], so it has not been possible to verify their results.

The most critical issue with *MID* is, however, the wrong implementation of one of its core components: the encoder of the variational autoencoder. As already discussed in Section 3.6, *MID* uses the *Trajectron++* encoder during data pre-processing to embed socio-temporal features inside the models; unfortunately though, an old implementation of it was being used, and this version suffered from a major bug as reported by an issue on the relative repository [32]. This bug consisted of a wrong computation of derivatives, which are used to calculate the velocity and the acceleration of pedestrians with regards to their space and time. In the affected code, the `np.gradient` method [33] was used, but this function applies a heuristic which is not suitable for this specific application domain: in fact, it computes the derivative from the positions at time steps $t - 1$ and $t + 1$ for the velocity, and the same goes for acceleration. This way, the outdated code was embedding future information into history training and test data, effectively cheating the model. The result from this bug was a lower result for *ADE* and *FDE* metrics; this could look as beneficial, but to a more careful

[1] Binary files containing the best-trained models that can be loaded into the code and enable almost full reproducibility of the results.

evaluation this is just unfair because the model knows information that is not supposed to know, making it to cheat to all effects.

All these problems together made the results reported in the original paper to lose credibility, as confirmed by the experimental results reported in Chapter 6. Moreover, a first marginal but fundamental fix had to be done, which was to fix the *Trajectron++* bug; as expected, the *ADE* and *FDE* results were higher on the same models. Therefore, all the results reported in this work refer to the corrected version of the encoder: even though we had significant improvements with respect to the baseline results, we were never able to achieve the same performances as the bug-affected version.

As a final note, we can say that the provided code came with almost no comment at all: therefore, part of the initial effort has been to write a documentation for the entire project; this has been developed with *handsdown* and *mkdocs* python libraries and is freely available on the same repository of the project.

5.2 Improving the existing model

Improving the existing model comprised to adapt the observations reported in *IDDPM* from synthetic image generation to trajectory generation and prediction. These consist of three main improvements plus an ablation study which was considered to be possibly useful and applicable to the trajectory forecasting domain. As extensively reported in Chapter 6, two of these implementations led to actual improvements in the model's performances, while two of them were not able to improve the results. However, we decided to report all of them because of the effort they took and because it is still useful to know what could or could not work; moreover, a sketch of the proof for which they don't bring any experimental improvements is provided in Chapter 6.

5.2.1 Noise schedule

The first step in improving the existing model has been to modify the noise schedule. As reported in Section 3.5.1, the authors of *DDPM* found it beneficial to apply a linear noise schedule to the forward diffusion process; the authors of *IDDPM*, on the other hand, found that this schedule worked well for high-resolution images, but was sub-optimal for lower res-

olution ones like 64x64 and 32x32. The reason for this phenomenon is that the forward process was too noisy close to the last steps of the corresponding diffusion: this led to a hard-to-train reverse process, and thus to a lower sample quality.

To address this problem, the authors propose a cosine schedule. Instead of a linear amount of noise, they construct a new cosine schedule defined as follows:

$$\beta_t = 1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}} \tag{5.1}$$

where β_t are the variances and $\bar{\alpha}_t$ is equal to

$$\bar{\alpha}_t = \frac{f(t)}{f(0)} \tag{5.2}$$

In this equation, $f(t)$ is the actual cosine noise schedule and, in their actual setting, it is defined as follows:

$$f(t) = \cos\left(\frac{\frac{t}{T} + s}{1 + s} \cdot \frac{\pi}{2}\right)^2 \tag{5.3}$$

where T is the maximum amount of diffusion steps and s is a small offset that prevents β_t from being too small near the first forward diffusion step. To be more specific, in their setting they found it beneficial to set $s = 0.008$, which resulted from the calculation of their pixel bin size. Their idea is then to apply these values to the variance calculations during the forward and the reverse process; this could also be applied to our domain, hence we implemented it.

The development of this iteration started with implementing the cosine noise schedule as defined in Equation 5.3; unfortunately, the first results were far from optimal, thus more experiments were needed. The core concept of this non-linear noise schedule was to find a function which has a linear-like behavior in the middle of the diffusion processes while changing little near the extremes (hence at $t = 0$ and $t = T$); this way, the data distribution is not totally corrupted at the end of the forward process, but it still has a reasonable amount of noise so that the model can be effectively trained.

With these considerations in mind, we developed two functions which could correspond to these properties: a modified version of the original cosine and a sigmoid-based function. Other functions such as tangents and a piece-wise function have been considered, but their results were poor and we consider them to be not particularly interesting, thus they are not reported. Both these functions are included in the `VarianceSchedule` class `[diffusion.py]`.

The modified version of the cosine schedule is defined by the following function:

$$f(t) = \cos\left(\frac{\frac{t}{T} + s}{1 + s} \cdot \frac{2}{5}\pi\right)^2 \tag{5.4}$$

As the reader may notice, this is fundamentally similar to the original function: in fact, after many experiments, we acknowledged that a simple re-scaling factor could work. However, this function resulted to bring the best results only on some datasets, as reported in Chapter 6; on the other datasets, the sigmoid function was observed to perform better. This function is defined as follows:

$$f(t) = \frac{1}{1 + e^{-\frac{3}{5}(x-\epsilon)}} \tag{5.5}$$

These two functions are used to compute different schedules by computing β_t using Equations 5.1 and 5.2; the schedules can be visualized in Figure 5.1.

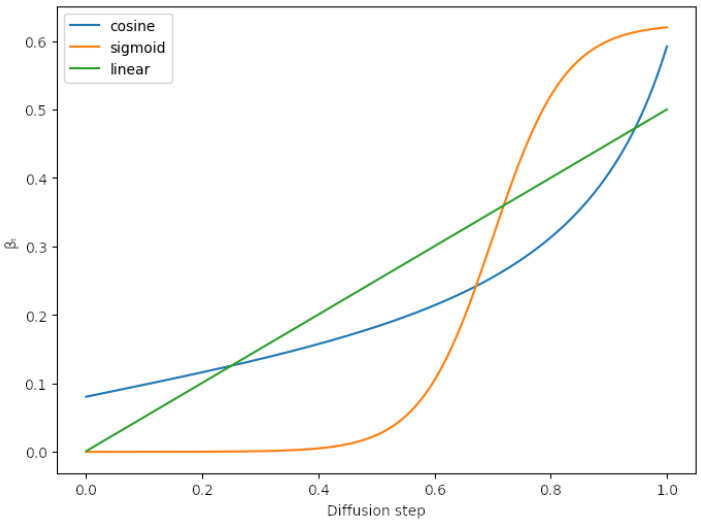

Figure 5.1: Noise schedules compared: linear, cosine, sigmoid.

As the reader may notice, only one of the two newly-defined non-linear functions totally behaves as the authors of *IDDPM* suggest: the sigmoid-based one, indeed, applies very little noise at the beginning and the end of the diffusion steps, while the cosine-based one applies noise more gradually at the beginning but it increases at the end of the noising process. However, as the results reported in Chapter 6 confirm, the latter works experimentally quite well; the reason behind this lies in the fact that, being the amount of noise applied lower than the linear for most of the steps, the total final amount of noise is still lower than with the linear schedule, thus it still enables the model to remove it and, therefore, learn suitable parameters.

5.2.2 Learning the variance

The second step in improving *MID* has been to enable the model to learn the variances. As described in Section 3.5.1, the authors of *DDPM* decided to just learn the means by fixing the variances σ_t to the noise schedule β_t; in their opinion, in fact, learning variances should not allow for better sample quality, thus the effect of learning them would only be to slow down the training process. Of the same idea are the authors of *IDDPM* but with one, paramount difference: they claim that it is true that fixing σ_t is a reasonable choice when talking of sample quality, but the same does not hold for log-likelihood. Indeed, they proved that the first steps of the diffusion process are the most important when optimizing the variance lower bound, thus the likelihood could be improved by choosing a better schedule for the variance. Considering this, they claim that we should learn them to obtain this improvement.

Network expansion

The first step has been to extend the *MID* network: as extensively described in Section 3.6 the core neural network, which is the Transformer based decoder depicted in Figure 3.11, takes as input the corrupted data $\mathbf{y}_k$, the state embedding and the time embedding, and it generates a two-dimensional output; this consists in a tuple (μ_x, μ_y) which represents the means for the bivariate Gaussian distribution which models the predicted noise. In order to learn the variance, we needed an additional output that could represent the variances inside the decoder. Hence, we simply modified the output layer so that it could output four values: the two components of the mean and a two-dimensional vector $\mathbf{v}$ which could be subsequently used to compute the variances.

Since the last layer of the decoder is a linear layer, it has been quite straightforward to make it output two more values: we just increased its number of nodes and we considered the first two for the mean and the last two for the variance. Practically speaking, we modified the `forward` method of the main Transformer class, which in *MID* is called `TransformerConcatLinear [diffusion.py]` to output the information needed.

Variance calculation

In theory, outputting two more values could be enough: the values from these nodes could be considered as variances as-is; however, as the authors of *IDDPM* point out, the reasonable range for the variance is very small, thus it would be a very hard task for the network to figure it out and re-scale it accordingly. Instead, they found it beneficial to parameterize the variance as a logarithmic interpolation between β_t and $\tilde{\beta}_t$; the variances are then calculated as follows:

$$\Sigma_\theta\left(x_t, t\right) = \exp\left(\mathbf{v}\log\beta_t + (1-\mathbf{v})\log\tilde{\beta}_t\right) \tag{5.6}$$

where $\mathbf{v}$ is the aforementioned two-dimensional vector.

Practically speaking, we extended the `VarianceSchedule [diffusion.py]` class to compute the parameters we need in order to compute the variance itself; originally, this class was meant as a mere container of the noise schedules with some additional utility method, while now it is more sophisticated. In particular, before the training phase, we compute the two coefficients $\boldsymbol{\beta}$ and $\tilde{\boldsymbol{\beta}}$ which are the vectors composed by, respectively, β_t and $\tilde{\beta}_t$ for each diffusion step t. The former's elements are calculated as follows:

$$\beta_t = 1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}} \tag{5.7}$$

with

$$\bar{\alpha}_t = \frac{f(t)}{f(0)} \tag{5.8}$$

where the function $f(t)$ is the noise schedule of choice, as reported in the previous paragraph.

Conversely, the latter is directly computed as a vector as follows:

$$\tilde{\boldsymbol{\beta}} = \log\left[\frac{\boldsymbol{\beta}[1-\Pi^{-1}(1-\boldsymbol{\beta})]}{1-\Pi(1-\boldsymbol{\beta})}\right] \tag{5.9}$$

where Π is intended as the cumulative product with the same size as the original vector and in which the i-th element is the product of all the elements from 0 to i, and Π^{-1} is the same vector but flipped.

Using these two vectors, the variance is then calculated for the correspondent noise step during the `forward` process of the decoder, thus during training, by applying Equation 5.6 almost as-is; the only difference is that we multiply the vector $\mathbf{v}$ by $\frac{1}{2}$ for normalization. Practically speaking, these operations are performed by the `get_log_sigmas_learning` method in the `VarianceSchedule` class; the only information needed (*i.e.* input parameter), besides the obvious $\mathbf{v}$ vector, is the diffusion timestep t. Before training, two more components are calculated for the subsequent loss function calculation: these are called as c_0 and c_1, and they are the coefficients for the posterior mean. They are defined as follows:

$$\begin{aligned} c_0 &= \frac{\boldsymbol{\beta}\sqrt{\Pi^{-1}(1-\boldsymbol{\beta})}}{1-\Pi(1-\boldsymbol{\beta})} \\ c_1 &= \frac{[1-\Pi^{-1}(1-\boldsymbol{\beta})]\sqrt{1-\boldsymbol{\beta}}}{1-\Pi(1-\boldsymbol{\beta})} \end{aligned} \tag{5.10}$$

Training

In order to learn the variances, a new loss function must be defined; the former L_{simple}, in fact, does not include the parameters for variance learning. The authors of *IDDPM* solved this problem by implementing the variance lower bound already described in Equation 3.29; this loss is so composed:

$$L_{vlb} = \begin{cases} -\log p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right) & \text{if } t = 0 \\ D_{KL}\left(q\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0\right) \| p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)\right) & \text{if } t > 0 \end{cases} \tag{5.11}$$

The first term is the negative log-likelihood of the first diffusion step; in practice, this is heuristically computed by the following:

$$\begin{aligned} -p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right) = \frac{1}{\log 2}\mathbb{E}\Bigg[&\frac{1}{2}\operatorname{cdf}\left(e^{\boldsymbol{\sigma}/2}(\mathbf{x}_0 - \boldsymbol{\mu} + \frac{1}{255})\right) \\ &- \frac{1}{2}\operatorname{cdf}\left(e^{\boldsymbol{\sigma}/2}(\mathbf{x}_0 - \boldsymbol{\mu} - \frac{1}{255})\right)\Bigg] \end{aligned} \tag{5.12}$$

where $\mathbf{x}$ are the ground truth future trajectories, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are the vectors of, respectively, the

predicted means and variances and cdf is another heuristic which computes the cumulative distribution function in a fast fashion.

The second term of the variance lower bound is computed directly the following:

$$\begin{aligned} & D_{KL}\left(q\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0\right) \| p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)\right) \\ = & \frac{1}{2\log(2)}\mathbb{E}\left[\left((\boldsymbol{\sigma} - \tilde{\boldsymbol{\beta}}) + e^{\tilde{\boldsymbol{\beta}}-\boldsymbol{\sigma}} + (c_0\mathbf{x}_0 + c_1\mathbf{x}_t - \boldsymbol{\mu})^2 e^{-\boldsymbol{\sigma}} - 1\right)\right] \end{aligned} \tag{5.13}$$

where $\mathbf{x}_t$ is the predicted future trajectory at timestep t and c_0 and c_1 are the coefficients listed in Equation 5.10.

Theoretically speaking, optimizing L_{vlb} should allow learning both the means and the variances of the final probability distribution; however, as the authors noticed and as discussed in Sections 5.2.3 and 5.2.4, this function is hard to optimize. Being the objective very noisy, and since we only need to learn the variances alongside the means which are already efficiently computed by L_{simple}, the authors found it beneficial to optimize a hybrid loss:

$$L_{hybrid} = L_{simple} + \lambda L_{vlb} \tag{5.14}$$

where λ is a coefficient used to regulate the learning balance between the two losses. In particular, the authors of *IDDPM* set it to $\lambda = 0.001$. As better described in Chapter 6, we also went through this problem when optimizing L_{vlb}, thus we adopted the same strategy as *IDDPM*.

Sampling

The sampling procedure for this new version of *MID* is almost the same as the one listed in Algorithm 3.2; however, since the variances are no more fixed, we need to consider them also during the generation of the trajectories. In order to do so, at each sampling step we compute the variances with Equation 5.6 starting again from the vector v. Then, for each diffusion step, we compute the trajectories using the following:

$$\mathbf{y}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{y}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_{(\theta,\psi)}\left(\mathbf{y}_t, t, \mathbf{x}\right)\right) + e^{\frac{1}{2}\log\sigma_\theta}\mathbf{z} \tag{5.15}$$

where σ_θ is the learned variance. The final vector $\mathbf{y}_0$ contains the final predicted trajectory.

5.2.3 Reducing gradient noise

As it should be clear, the variance lower bound is a hard-to-optimize function; this very consideration is also reported in *IDDPM*, where the authors claim they expected to achieve better likelihoods by directly optimizing L_{vlb} but, due to the noisy gradient, the results were just poor. In order to address this problem, they propose a re-scaling technique based on importance sampling; this is defined as follows:

$$L_{vlb} = E_{t \sim p_t} \left[\frac{L_t}{p_t} \right] \tag{5.16}$$

where

$$\begin{aligned} p_t &\propto \sqrt{E[L_t^2]} \\ \sum p_t &= 1 \end{aligned} \tag{5.17}$$

The intuition behind this formula is that we need to rescale each component of the loss by some weight; with *component of the loss* we mean the batch loss on one diffusion step, and the weight is computed at run time according to the mean of the losses of some pre-defined number of diffusion steps. In particular, since the mean $E[L_t^2]$ could (and should) change during the training, the authors of *IDDPM* found it beneficial to set this window size to ten; that is, the rescaling factor p_t is the mean of the loss components of the previous ten diffusion steps. Moreover, since at the beginning of the training phase the losses L_t are unknown, they start applying the importance sampling after the first ten training iterations of each epoch.

Practically speaking, this technique has been introduced in this work by implementing the `LossSecondMomentResampler` class `[resample.py]`; this has the purpose of memorizing the loss history and computing the weights for the subsequent losses accordingly. This class is instantiated before the beginning of the training phase, hence we can easily access all the needed information at run time. For each batch in each epoch starting from the tenth iteration on, the weights for each loss and their respective indexes are sampled by the `sample` method within the resampler class. The weights are then updated with regards to the local loss by the `update_with_local_losses` method, and they are finally used for importance sampling by multiplying the losses by them. The losses are then averaged to obtain the final loss to optimize.

With regards to the `sample` method, this works as follows: first of all, it calls the `weights` method to compute the weights p_t; then, it uses these values to compute the probabilities of each timestep to be chosen for the importance sampling. The `weights` method, in turn, checks if the sampling is warmed up, *i.e.* the iteration number is greater than ten, and computes the weights p_t according to Equation 5.17; specifically:

$$p_t = \frac{\sqrt{E[L_t^2]}}{\sum_t E[L_t^2]} \cdot (1 - u) + \frac{u}{t} \tag{5.18}$$

where u is the probability density function of a uniform distribution and is experimentally fixed to $u = 0.001$.

Still in the `sample` method, the probabilities are calculated as $\frac{p_t}{\sum_t p_t}$, and these are used to choose the diffusion steps to which apply the importance sampling. The method finally returns these timesteps and the associated weights.
On the other hand, the `update_with_local_losses` updates the re-weighting by using the losses from the model. Its functioning is straightforward: at each call, it shifts the oldest loss term and just updates the losses window; the further weight updates are performed by the two other aforementioned methods.

As better analyzed in Chapter 6, gradient noise reduction has been implemented and tested for two different losses: L_{vlb} and L_{hybrid}. The authors of *IDDPM* did not find it beneficial to apply it to the hybrid objective, but we experimented also with this in order to check if it could apply to the different task of trajectory prediction.

5.2.4 Loss ensemble

As better described in Chapter 6, directly optimizing L_{vlb} by reducing the gradient noise did not bring any considerable result: conversely, we found this version of the model to be the worst. Consequently, we decided to implement an ablation from the *IDDPM* paper, the loss ensemble, which could improve in this framework.

The main idea lies in the fact that both L_{hybrid} and L_{vlb} have their advantages and their downsides: in particular, the former has been seen to focus more on the general context of the

generated data, while the latter tends to focus more on the imperceptible details, neglecting the general scene; this is also confirmed by the explanation for L_{vlb} utilization we gave in the previous section. The intuition the authors of *IDDPM* had, then, is to use both losses, or better one loss at a time for particular time ranges. In particular, they found that the model resulting by optimizing the variance lower bound is better at the start and the end of the diffusion process, while the one resulting from the hybrid objective returns higher quality samples in the middle of the diffusion process.

In order to address this intuition, we performed a little implementation of the loss ensemble. In particular, we defined a new loss function as follows:

$$L_{ensemble} = \begin{cases} L_{hybrid} & \text{if } t \in [N, T-N] \\ L_{vlb} & \text{elsewhere} \end{cases} \tag{5.19}$$

where t is the diffusion step, T is the number of diffusion steps and N is an empirically-found parameter that regulates when to use one or the other loss function.

Practically speaking, we just updated `DiffusionTraj [diffusion.py]`, which is the class that manages the diffusion processes, so that it could compute the right loss for the specific diffusion step t; the two losses had already been implemented during the variance learning iteration, thus it has been straightforward to perform this task. In particular, since the authors of *IDDPM* set $N = 100$ on total diffusion steps of $T = 1000$, we set $N = 10$ since in our framework $T = 100$. The corresponding loss is then optimized in the same way of L_{simple}, L_{hybrid}, and L_{vlb}.

5.3 Implementing the goal module

The last considerable iteration for the current work has been to merge *MID* with *Goal-SAR* by implementing a goal module into the former. As we will deeply explain in this section, however, we started with this idea in mind but then we went further by implementing also the self-attentive recurrent backbone; after some trial and error, we finally got to a version which could possibly take the best from the two models, hence bringing a tangible improvement.

This model's architecture is depicted in Figure 5.2, of which the next sections will explain the composition. Before starting to analyze it, it is important to notice that this new architecture is composed of two different networks which are trained at different times: the first is the self-attentive recurrent backbone, which is pre-trained from the whole trajectory information and the final goal and its parameters are saved. The second network is the standard *MID* with the addition of the goal module; this also samples from the pre-trained *SAR* network, and it combines the results from the two networks by averaging them.

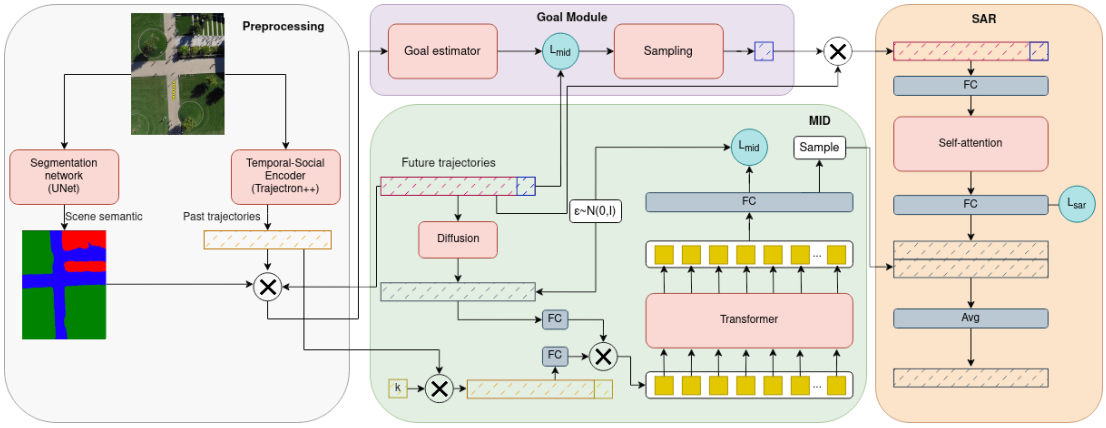

Figure 5.2: Architecture of MID with the goal module and the recurrent self-attentive network.

5.3.1 Pre-processing

This iteration started with thinking and implementing the leftmost square of the model's architecture, which represents the pre-processing procedure; the two works' structures are, in fact, very different both in terms of which data is used and how this data is loaded for it to be used from the model.

First of all, in *MID* we didn't have either the RGB image or the scene semantic; while the former is not useful in terms of model training but only for results visualization, the latter is a fundamental part of the goal module, thus we needed to find a way to load it into the model. Practically speaking, we acted on several fronts: first of all, we created two new classes which could represent this information which are called, respectively, `Map` and `SemanticMap [map.py]`. The former comprises the RGB image of the scene as a numpy

array and its homography[2], which is used to map the points from real world to pixel coordinates (and vice versa by inverting the matrix); the class has then a public `translate_trajectories` method which can apply the homography. On the other hand, the latter is composed of two fields, which are the raw data and the tensor image; this last field is computed by the `_create_tensor_image` method, which creates the stack of semantic layers described in Section 3.4. In particular, this stack is directly computed at the time when the class is instantiated, and it is composed of six layers, one for each semantic class: *unlabeled*, *pavement*, *road*, *structure*, *terrain* and *tree*.

After having implemented the maps classes, we moved on to expand the `Scene` class `[scene.py]` in order for it to contain the maps; this has been easily performed by just adding two more fields, one for map type. These fields are instantiated during the pre-processing procedure, along with the usual data. Moreover, even though we were already loading the whole trajectories (history included), the way *Goal-SAR* is thought of is different: as better described in Section 3.4, the trajectories are represented as a stack of probability maps where each map represents, in turn, one point in the space. Therefore, we needed to adapt *MID* code to work on this framework, since this representation is fundamental for its usage with semantic maps. Fundamentally, we modified the `Node` class `[node.py]` such that it contains this information; this, in turn, is computed during pre-processing by the `get_input_traj_maps` function. Moreover, we expanded the class by adding a getter method for the trajectory maps, so that these can be easily accessed during training and sampling.

With regard to the online pre-processing performed in `[preprocessing.py]`, we modified the related utilities so that we can have the needed information for the goal. To be more specific, we implemented a getter method which receives as input a timesteps range and a scene and returns the associated semantic map and the trajectory stack of maps.

5.3.2 Goal module

Now that we had all the needed data we started by implementing the goal module, which is represented by the top square in the architecture schema. This is composed by two main components: the goal estimator and the sampling phase; both of these have been imple-

[2] Formally, a homography is described as an isomorphism that maps points in space to points in another space. In practice, it is composed of a 3 x 3 matrix (in the case of an RGB image) to which a point is multiplied in order to obtain a diverse space representation of the point.

mented inside the very Transformer-based decoder of *MID*. The functioning of this component is the following: during the forward pass of the diffusion network, we concatenate the tensor semantic maps to the observed trajectory maps and we pass the resulting tensor as the input of the goal network. The latter is created at the moment of instantiation and consists in the same U-Net-based model of *Goal-SAR*, which has been already described in Section 3.4 [unet.py].

In total, we return three outputs, which are auxiliary to the already existing output of the diffusion network. The first and second values we return are the logit probability map of the goal, which is the direct output of the U-Net, and the ground truth map; both of these are used to compute the loss of the goal sub-network. Talking about the loss, this is computed together with the standard MSE-based loss of *MID* and it is the binary cross entropy loss between the two maps. The third and most important value we return is the actual goal point; this is different depending on which phase we are in: during the pre-training of the *SAR* network we return the ground truth goal, *i.e.* the last point of the future trajectory since our scope is to train only the recurrent network; when running *MID* to train the main model, on the other hand, we return the sampled goal. With regards to this, it is computed by the `sampling` function which receives the probability map, which in turn is the output of the U-Net to which we apply a sigmoid function, and returns the sampled map, which we convert to obtain the final goal coordinates.

Finally, the goal point is passed to the *SAR* network both when we are pre-training it and when we are using it to sample trajectories. This is better explained in the next sections.

5.3.3 SAR

As previously outlined, we decided to implement two models so that they can operate at different times. The reasons for this choice are many: first of all, we needed to implement a recurrent backbone to solve the dimensionality problem we had by merging two different models. *MID*'s diffusion network, in fact, outputs parameters for a normal distribution, while the main network from *Goal-SAR* directly outputs trajectories; hence, the two solutions we had were to sample from the distribution at training time or to implement another network which could use the goal information to predict the trajectories.

We decided on the latter mainly for performance reasons: sampling, in fact, is a very demanding task and, since in the former scenario we would have to sample for each iteration, this was just unfeasible. The reason why we chose to pre-train the model follows the same logic: we just need to pre-train the *SAR* network once, and then we can use it for many different experiments. Moreover, training *SAR* at the same time would mean adding another loss component to an already complicated objective. Finally, we would like to point out that the *SAR* network was not our first choice: before implementing it, in fact, we tried a simple recurrent network with a light attention mechanism; this, however, was too simple, hence the results were not optimal.

From the practical point of view, we adapted the self-attentive recurrent backbone already described in Section 3.4 to *MID* by removing the goal module: we have now a recurrent model which receives as input the history trajectory and the goal (which can be either the predicted one or the ground truth, as explained previously in this chapter) and outputs the future trajectory as a tensor of points in the space. These are then confronted with the ground truth future trajectory and the loss for the model is computed by minimizing the mean squared error between these two.

Since the network structure is essentially the same as *Goal-SAR*, we skip the detailed description; however, we need to make two considerations. The first is that the goal information is not computed at run-time, but it is given by the ground truth while pre-training and by the sampling from the diffusion network when training and sampling the *MID* model. Moreover, the output from the *SAR* network is not used as-is: as the architecture schema suggests, in fact, we combine it with the output from the sampling procedure of *MID* by applying a point-wise average.

5.3.4 Training and sampling

Now that we have explained all the single components, it is time to wrap up and explain how the pipeline for training and sampling works. A typical execution of the model starts now with the pre-training of the *SAR* network: as already described, this takes as input history trajectory and goal point and outputs the future trajectory; at the end of this training, the model parameters are saved for the subsequent use. After this, the *MID* network is trained in the usual way: during this phase, the goal module inside the Transformer decoder is also

trained as described in Section 5.3.2. When both the *SAR* and the *MID* models are trained, they can be used to sample the trajectories; in order to do so, we sample the trajectories from *MID* in the usual way, and we use the *SAR* network to obtain another trajectory representation, which is then point-wise averaged to the original one. As a last note, it is important to notice that, in order to not introduce bias in the *SAR* model, the goal information we feed it is the sampled one, as opposed to the ground truth one used for training.

5.4 Long-term prediction

As mentioned in the introduction to this work, as a side task we decided to modify the *MID* project in order to adapt it to the long-term prediction scenario. The main reason behind this choice is that we wanted to test this model on this framework, and evaluate its performances with respect to the results of Y-Net, which we cited in Section 2.3.3 and is both the first work and the state-of-the-art when talking of long-term prediction.

We already explained what long-term prediction means at the beginning of Chapter 3: in short, we want to predict the goal and the path a pedestrian is taking for a longer time, in particular, we want to generate the predictions of $T_{pred} = 30s$ after $T_{obs} = 5s$ of motion history. The reasons why studying this scenario is important are various: first of all, the long-term task is enormously harder than the short one, thus we can try to optimize it in order to optimize the easier objective. Moreover, the results from a long-term prediction could give useful insights to understand why a model behaves in a way or another.

Concretely speaking, in order to adapt *MID* for this framework we had to act on different fronts, which are described hereby. First of all, we had to modify the data pre-processing procedure; the most notable difference lies in the dataset, which consists of a different version of the Stanford Drone Dataset: both the original one and the ETH/UCY datasets, in fact, don't have long enough trajectories. This data has been picked up from the Y-Net repository [6], and the code for pre-processing has been adapted to its formatting; moreover, we also needed to change the timestep size dt according to how the data has been registered.
Secondly, we needed to modify the *Trajectron++* parameters and the training procedure; with regards to the former, we just needed to change the number of both history and future timesteps, which are called respectively *minimum history length* and *prediction horizon*. Then, we adapted the training procedure so that it could feed the model the right amount

of timesteps; the same goes for the sampling procedure, where we need to generate longer future trajectories.
Finally, we slightly modified the Transformer-based decoder: the positional encoding, in fact, could only accept trajectories with less than 24 steps, while we needed at most 30; this, however, has been an easy task since we could just modify one hyperparameter.

6

Experiments and results

In this chapter we provide the most important results from the several experiments we portrayed during this thesis work: in the first section we report the best results we got overall, and in the second section we ablate these results and the others. Before starting with showing the results, we remember that we base our assumptions and considerations on mainly two metrics, which are the *ADE* and *FDE* described in Section 3.1.2, since most of the literature on the trajectory prediction field only uses these two; however, for the ablations we report also the *KDE-NLL* metric since it is fundamental to describe the phenomena we went through the experimentation.

All the experiments have been performed with the Python interpreter set to version 3.10. The main hardware we used is either a Tesla V100 or a Quadro M400 GPU; however, due to the fact that most of the diffusion calculations need to be performed on the CPU, we noticed little differences in terms of training and sampling speed between the two graphic accelerators. As a final note, we would like to point out that we implemented a visualization and metrics reporting tool, based on *wandb* [34], in order to have qualitative results on the predictions; some of these are reported in the sections below.

6.1 Summary

In this section we report the results we got from the best model, which turned out to be the one with only the different noise schedule; however, as pointed out in the introduction to this chapter, this consideration holds only in terms of *ADE* and *FDE*, thus we are purposely ignoring the trajectory prediction as a whole and we are considering only the best-out-of-20 samples at each sampling step. The results are reported in Tables 6.1 and 6.2.

Method	ETH	HOTEL	UNIV	ZARA1	ZARA2	AVG
Social-LSTM	0.50/1.07	0.11/0.23	0.27/0.77	0.22/0.48	0.25/0.50	0.27/0.61
Trajectron	0.40/0.78	0.19/0.34	0.47/0.98	0.32/0.64	0.33/0.65	0.34/0.68
Trajectron++	0.39/0.83	0.12/0.21	**0.20/0.44**	**0.15/0.33**	**0.11/0.25**	0.19/0.41
PECNet	0.54/0.87	0.18/0.24	0.35/0.60	0.22/0.39	0.17/0.30	0.29/0.48
Goal-GAN	0.59/1.18	0.19/0.35	0.60/1.19	0.43/0.87	0.32/0.65	0.43/0.85
Y-Net	**0.28/0.33**	**0.10/0.14**	0.24/0.41	0.17/0.27	0.13/0.22	**0.18/0.27**
Goal-SAR	**0.28/0.39**	0.12/0.17	0.25/0.43	0.17/0.26	0.15/0.22	0.19/0.29
MID - baseline	0.54/0.82	0.21/0.32	0.31/0.56	0.28/0.51	0.20/0.36	0.31/0.52
MID - ours	0.49/0.77	0.17/0.27	0.29/0.53	0.24/0.44	0.19/0.35	0.28/0.47

Table 6.1: ETH/UCY best results. For each dataset and each model, we report respectively $min_{20}ADE$ and $min_{20}FDE$. Bold and underlined results are respectively the best and the second best result for the dataset (lower is better).

Method	SDD
Social-LSTM	57.00/31.20
PECNet	9.96/15.88
Goal-GAN	12.20/22.10
Y-Net	7.85/11.85
Goal-SAR	**7.75/11.83**
MID - baseline	10.70/18.21
MID - ours	9.95/16.69

Table 6.2: SDD best results. For each model we report respectively $min_{20}ADE$ and $min_{20}FDE$. Bold and underlined results are respectively the best and the second best result for the dataset (lower is better).

As a side note, as anticipated in Chapter 5, different noise schedules behave in differ-

ent ways with respect to the different datasets. In particular, we found it beneficial to use the sigmoid schedule reported in Equation 5.5 to train the model on the *eth* subset of the ETH/UCY dataset, while we used the cosine schedule represented by Equation 5.4 for the others.

It is important to notice that the best results we managed to achieve are still far from what was reported in the original *MID* paper; this matter has been better analyzed in the first section of Chapter 5, and is mainly caused by the wrong implementation of *Trajectron++*. However, we consider them to be the best since they significantly outperform the best results we got on the baseline code, which are reported in the table above. For reference, the reported results from *MID* original paper are listed in Tables 6.3 and 6.4.

Method	**ETH**	**HOTEL**	**UNIV**	**ZARA1**	**ZARA2**	**AVG**
MID - original	0.39/0.66	0.13/0.22	0.22/0.45	0.17/0.30	0.13/0.27	0.21/0.38

Table 6.3: ETH/UCY original MID results. For each dataset and each model we report respectively $min_{20}ADE$ and $min_{20}FDE$. Bold and underlined results are respectively the best and the second best result for the dataset (lower is better).

Method	**SDD**
MID - original	7.61/14.30

Table 6.4: SDD original MID results. For each model we report respectively $min_{20}ADE$ and $min_{20}FDE$. Bold and underlined results are respectively the best and the second best result for the dataset (lower is better).

We would like to point out that even the original version of *MID* has not been able to outperform other consolidated works such as *Y-Net*, *Trajectron++* and *Goal-SAR*; however, as we discuss in the conclusion to this work, the use of diffusion models for the trajectory prediction task is only in its infancy, thus there is hope for improvement with further research.

As a final note to these results, we would like to mention that diffusion models show great potential in training time when talking about performance. Even though other generative approaches such as Generative Adversarial Networks (GANs) and Variational Autoencoders have been seen to produce samples in a lower computational time [35], [36], they usually take a great amount of time and huge amounts of data in order to effectively get trained. Diffusion models, contrariwise, tend to have two useful properties: they need usually less

time and less data to get trained, as [36] suggests, and they are less prone to overfitting with regard to other models. Even though this is not a formal proof, Figure 6.1 shows this exact behavior: the two precision metrics *ADE* and *FDE* tend to quickly decrease at the very beginning of training; then, they continue to decrease in a slow fashion and, most importantly, they difficultly explode when overtraining.

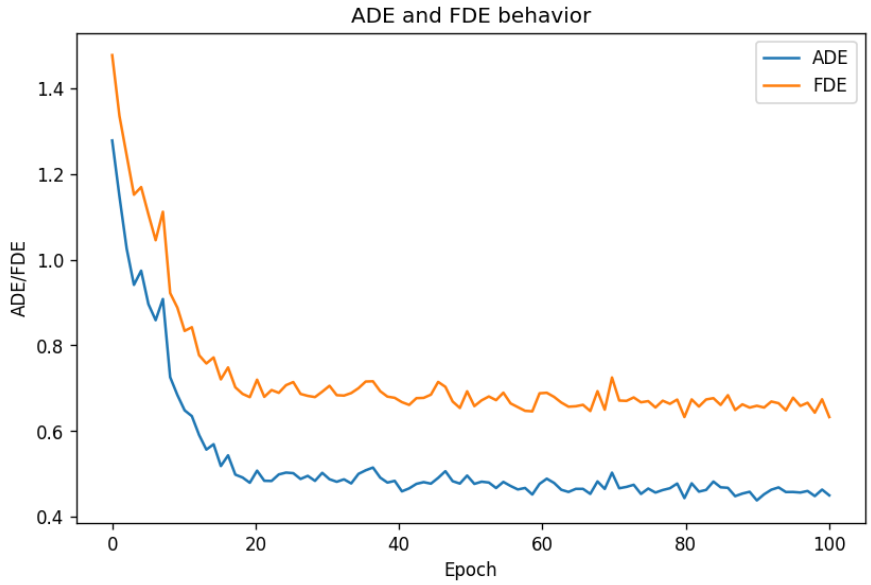

Figure 6.1: Training speed of MID in terms of ADE and FDE. Both metrics are calculated at each epoch on a validation set.

6.2 Ablations and considerations

In this section we will present some considerations and some alternative results we got on the different phases of this project. Of paramount importance is the variance learning section: this implementation, in fact, brought just little improvements in terms of *ADE* and *FDE*, but when taking into consideration also the distribution of the generated trajectories, this becomes fundamental.

6.2.1 Noise schedule

As anticipated at the beginning of Chapter 5, implementing the non-linear noise schedule enabled an improvement in the performance of the model; indeed, to be precise, this implementation brought the best results in terms of $min_{20}ADE$ and $min_{20}FDE$. The reason

for this lies in the dimensionality of the problem: having the trajectory prediction task a substantially lower dimensionality of the synthetic image generation one, the considerations the authors of *IDDPM* made on image dimensions are more than ever true. For this reason, a noise schedule different from the linear one will probably always be the best choice for diffusion models applied to this framework, regardless of other improvements. This has also been verified by the subsequent experiments we tried on the other implementations: the linear schedule just does not work well when compared with the cosine or the sigmoid one. For comparison, Tables 6.5 and 6.6 show how the different schedules influence the results on the different datasets.

Method	ETH	HOTEL	UNIV	ZARA1	ZARA2	AVG
MID - linear	0.54/0.82	**0.18/0.27**	0.32/0.59	0.28/0.50	0.20/0.36	0.30/0.51
MID - cosine	0.52/0.84	0.20/0.30	**0.29/0.53**	**0.24/0.44**	**0.19/0.35**	**0.28/0.49**
MID - sigmoid	**0.49/0.77**	0.21/0.32	0.33/0.67	0.26/0.45	0.20/0.36	0.29/0.51

Table 6.5: ETH/UCY results with different noise schedule. For each model and each dataset, we report respectively $min_{20}ADE$ and $min_{20}FDE$. Bold results are the best for the dataset (lower is better).

Method	SDD
MID - linear	10.65/18.45
MID - cosine	**9.95/16.69**
MID - sigmoid	10.03/16.80

Table 6.6: SDD results with different noise schedules. For each model we report respectively $min_{20}ADE$ and $min_{20}FDE$. Bold results are the best for the dataset (lower is better).

6.2.2 Variance learning

As mentioned, learning the variance by optimizing L_{hybrid} brought only little improvements in terms of the distance metrics; as a matter of fact, it slightly outperforms the original *MID* baseline, but it cannot compete with the results of the noise schedule. However, it is very important to notice that it brings considerable results in terms of KDE-NLL; these are reported in Table 6.7.

Method	ETH	HOTEL	UNIV	ZARA1	ZARA2	AVG
Social-GAN	14.70	8.10	2.88	1.36	0.96	5.80
Trajectron	2.99	2.26	1.05	1.86	0.81	1.79
Trajectron++	**1.31**	**-1.94**	**-1.13**	**-1.41**	**-2.53**	**-1.14**
MID - baseline	1.9	<u>1.1</u>	0.37	0.85	-0.28	0.72
MID - ours	<u>1.6</u>	1.23	<u>0.05</u>	<u>0.7</u>	<u>-0.31</u>	<u>0.65</u>

Table 6.7: ETH/UCY KDE-NLL results. Bold and underlined results are respectively the best and the second best result for the dataset (lower is better).

Note that the results are only from the ETH/UCY dataset since there is no literature about SDD dataset. However, the results for this dataset for our experiments are reported in Table 6.8.

Method	SDD
MID - baseline	0.1
MID - ours	-0.05

Table 6.8: SDD KDE-NLL results. Lower is better.

Even though we did not achieve state-of-the-art results, our approach gives the second-best results when confronted with the most influential papers which use this metric. Intuitively this result suggests that, even though the best-out-of-20 predictions are in general better without learning the variance, the model obtained by optimizing L_{hybrid} tends to produce better results *in general*. More formally, we can say that adding the variance to the equation confirms what the authors of *IDDPM* reported: it does not lead to a better sample quality, but to a better likelihood. In the trajectory prediction case, a better likelihood represents more plausible trajectories. This is qualitatively confirmed by the plots of the trajectories, of which Figure 6.2 is a vivid example and the plots in Appendix A are more subtle ones: as one can see, the predictions are in general closer to the ground truth on the learned variance version, even if the best-out-of-20 could be more distant.

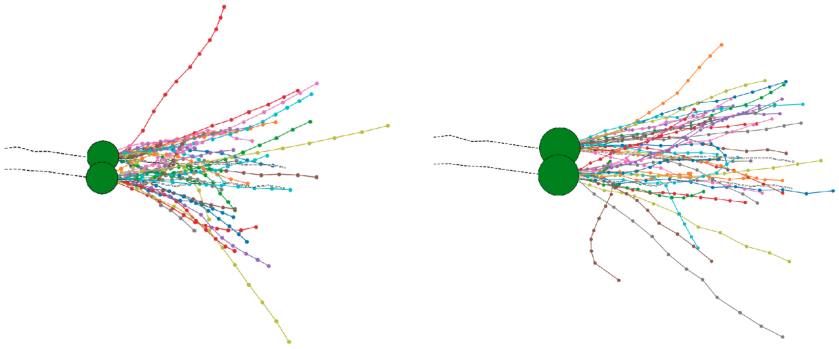

Figure 6.2: Comparison between a prediction on the ETH dataset using the non-learned version of the model (on the left) and the learned version (on the right). The dotted lines represent the histories of the two nodes, the green points are where the predictions start and the colorful segmented lines are the 20 predictions (for each agent).

6.2.3 Optimizing L_{vlb}

As hinted in the previous chapter, the two attempts in optimizing directly the variance lower bound, *i.e.* reducing the gradient noise via importance sampling and the loss ensemble, did not bring any tangible improvement; contrariwise, the results with these models are the worst ones. We decided to not report them, as not only they are not competitive with any other model (for instance, on *eth* subset dataset we got *ADE*/*FDE* metrics higher than, respectively, 4 and 5), but they also don't provide any additional information. However, we wanted to analyze here the possible causes of this phenomenon.

One possible cause can be traced to what the authors of *IDDPM* reported in their work: learning the variances can of course be beneficial to improve the likelihood, but they risk introducing too much spurious information inside the model. From another point of view, since learning them is not beneficial for the sample quality in synthetic image generation tasks, the same applies to the trajectory prediction task: indeed, this holds even more for the latter, due to its much lower dimensionality. This is still in line with *IDDPM* conclusions: the variance lower bound, in fact, tends to focus more on the imperceptible details rather than the big picture, and this makes the model lose generality; by focusing on small details, in fact, we risk to lose the recurrency property typical of trajectories.

Another possible cause for this phenomenon is that the L_{vlb} function is just too hard to optimize; this is a plausible scenario since it is also the assumption the authors of *IDDPM* made when they tried to implement the noise reduction via importance sampling. This can

be also the reason for the observations we made when we were trying to optimize $L_{ensemble}$: we noticed that the narrower the window in which we used L_{vlb} was, the better the results were.

In order to prove that our model was still working, but it just could not reach the same performances we had with different loss functions, we introduced a new loss as a new measurement. Originally, every loss we implemented for *MID* tried to compare the predicted noise with some pure Gaussian noise, thus it did not provide any information on the distance between the predicted and the ground truth trajectories, and it was naturally decreasing during training with every model's version. Therefore, we implemented a new MSE-based loss to measure this aspect; in order to not make it interfere with the training, we compute it by detaching every tensor and stopping the gradient. Then, we compared the trend of it with the trend of the usual L_{hybrid} by plotting them in Figure 6.3.

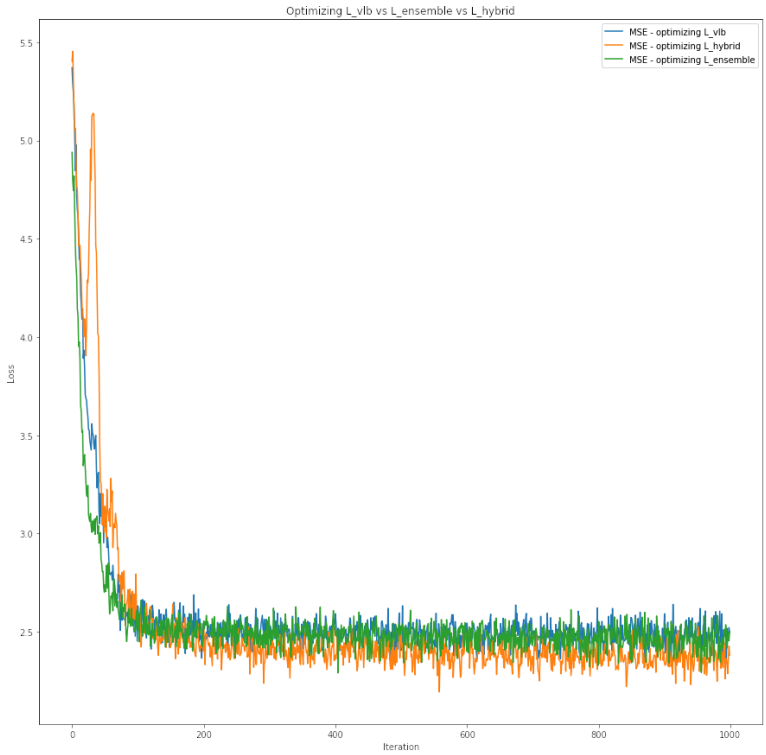

Figure 6.3: Comparison between the two variance lower bound based losses and the usual L_{hybrid}.

As the reader may notice, in all of the three models the loss is decreasing, but both L_{vlb} with importance sampling and $L_{ensemble}$ are not able to reach the performances of L_{hybrid}.

After this consideration, we concluded that learning the variance can indeed be beneficial, but we must do it in a way that does not interfere with the standard learning: L_{hybrid} is the shining example of this.

6.2.4 Goal module

An honorable mention in terms of results and possible improvements goes to the goal module implementation in *MID*. This model, in fact, has been able to outperform the original *MID*, but only on one subset in ETH/UCY; unfortunately, it was not able to reach the same precision for the others, hence we did not report it as the best. Moreover, it is fair to disclose that we only have the results on the ETH/UCY dataset because of computational reasons: the SDD dataset, in fact, is too big for the task and the actual hardware, as it is composed of the such high amount of scenes and pedestrians that its data does not fit entirely into the GPU memory. The results for ETH/UCY are reported in Table 6.9.

Method	**ETH**	**HOTEL**	**UNIV**	**ZARA1**	**ZARA2**	**AVG**
MID best	0.49/0.77	0.18/0.27	0.29/0.53	0.24/0.44	0.19/0.35	0.28/0.47
MID goal	0.44/0.65	0.32/0.56	0.44/0.78	0.29/0.55	0.25/0.45	0.35/0.60

Table 6.9: ETH/UCY results with the implemented goal module. For each model and each dataset we report respectively $min_{20}ADE$ and $min_{20}FDE$.

It is important to analyze some possible reasons why we were able to get better results for one dataset and worse for the others. The main reason behind this behavior is probably due to the different dimensions of the input data: when running on the *eth* subset of the ETH/UCY dataset, in fact, the model is being trained on scenes that are all of dimension 576x720 pixels, which are opportunely re-scaled by a constant, and its parameters are tested on the actual *eth* scene, which being 480x640 is the only different-sized scene in the dataset. When running on the other subsets, on the other hand, this difference in dimensionality needs to be taken into account, as we need same-dimensional scenes to create the stacks we described in Section 5.3.2. Therefore, to make the model work, we needed to upscale the *eth* scene to put it in the stack, and subsequently downscale it to produce the right output. It is therefore possible that, during this process, we lose some information, thus the reason for this performance drop.

Another source of inaccuracy is the way the results from the two networks *SAR* and *MID* are put together: the point-wise average is certainly a viable way, but it is probably not the best solution. As suggested in Section 7.1, in fact, a better solution could be to create a last auxiliary network with at least one linear layer. This network should take as input the two predicted trajectories and it should output the final one; moreover, it could be trained at the same time of *MID*, having the former a quite straightforward architecture.

Finally, it is important to mention that while training *MID* requires few epochs, and this network does not tend to overfit as mentioned in Section 6.1, the same does not apply for *SAR*. The latter network, in fact, needs a considerable amount of epochs to get properly trained due to its nature, *i.e.* it directly predicts trajectories instead of just Gaussian noise. Moreover, it is more prone to overfitting. Therefore, we are not sure that the *SAR* model we obtained during the experiments is the best one, thus we are not confident to say that the results we reported are the best the model can achieve. As discussed in Section 7.1, more experiments should be performed in order to assess this particular question.

6.2.5 Long-term prediction

In this section we report the results for the long-term prediction setting we talked about in Chapter 5; moreover, we compare them with the state-of-the-art, which is *Y-Net*, and with two more networks which were not meant to apply in this particular task, *Social-GAN* and *PECNet*. For clarity, we outline that we got these results from the *Y-Net* paper [6]. The comparison is shown in Table 6.10.

Method	**Keypoints**	**SDD**
Social-GAN	1	155.32/307.88
PECNet	1	72.22/118.13
Y-Net	1	47.94/66.71
	2	44.94/66.71
	5	39.49/66.71
MID (ours)	1	76.56/114.59

Table 6.10: Results for the long-term prediction task on the SDD dataset. For each model we report respectively $min_{20}ADE$ and $min_{20}FDE$ (lower is better). For Y-Net we report also the number of intermediate goals (keypoints).

In order to interpret these results we need to take into account two considerations: the first is that *MID* is not thought for this particular task, as well as *PECNet* and *Social-GAN*, and as reported in the correspondent section in Chapter 5 we just adapted the model to work in this framework without really trying to make it Taylor-made. The second is that *Y-Net*, being designed with this very setting in mind, both performs better than any other model and offers an additional feature, which is the number of intermediate goals we talked about in Chapter 2; no wonder, then, that *Y-Net* brings the best results.

However, it is interesting to note that *MID* performs quite well even in this out-of-the-box scenario: it definitely defeats *Social-GAN*, and it ties with *PECNet* (slightly higher *ADE*, slightly lower *FDE*); this suggests that the human indeterminacy removal postulated as the core principle of *MID* works also for harder tasks such as this. In Section 7.1 we give a hint about what the future work on this particular task could be to make the model perform even better.

7

Conclusion and future work

In this thesis, we studied the human trajectory prediction subject, which is a flagship topic at the moment of writing due to its important practical applications, such as autonomous driving and crowd surveillance. Most importantly, we deepened this topic from a fresh point of view, which is the one of considering the intrinsic human indeterminacy as noise we can try to remove to obtain plausible trajectories. Furthermore, it is interesting to note that diffusion models are acquiring more and more importance at the time of this report: *DALL-E 2* [37] and *Google Imagen* [38] are just two of the many emerging applications that, albeit different implementation techniques, fundamentally use this technology to produce convincing, high-quality samples.

To the best of our knowledge, *MID* is the first and only model that uses such diffusion approach applied to trajectory forecasting, and as we discussed in this work it needs some more work to become truly competitive with more established works like *Y-Net* and *Goal-SAR*. Nevertheless, we see a fair amount of room for improvement in the upcoming research. Some of this improvement might come from this very work, in which we applied the considerations given by the *IDDPM* paper to this context and, as discussed in the previous chapters, we were indeed able to outperform the original results we got on the project baseline, albeit we were not able to outperform the results the authors reported in their work due to the *Trajectron++* bug discussed in Chapter 6. Of these improvements, we would like to put a special focus on the improved noise schedule, which brought the best improvements in

terms of $min_{20}ADE$ and $min_{20}FDE$, and on learning the variance, which improved the general model's performance by significantly decreasing the negative log-likelihood.

Moreover, we think that further work is needed on the goal module implementation, as discussed in Section 7.1. The main advantages, for this particular task, of generating samples via diffusion processes are the considerable diversity of the outputs and the precise modeling of motion indeterminacy; it is our belief that this, together with some stronger and more information-inclusive models such as *Goal-SAR*, could concur to effectively produce trajectories that are plausible and precise.

7.1 Future Works

Even though we can be satisfied with the implementations we performed and the results we achieve during this work, there are still some improvements that can be done in order to possibly obtain even better results. Most of these possible future works regard the integration of the goal module: first of all, some experiments on the *SAR* training can be performed. As mentioned in Chapter 6, in fact, we are not sure that the models we obtained are the best, as *SAR* needs a considerable amount of training time to properly train; hence, future work could be to train the network for a longer time and/or with a lower learning rate in order to improve its performances and avoid overfitting. Talking about this, additional techniques such as other regularization techniques and early stopping can be adopted to reduce overfitting.

Another improvement could be to solve the dimensionality problem we experienced when training *MID-goal* on the ETH/UCY dataset: avoiding upsample and subsequently downsample should solve the performance issues we had on this dataset. Moreover, solving this problem should enable to pre-train *SAR* on all the subsets at once, improving its general performance. Finally, a last improvement in the goal module implementation could be to change the way the results from the two networks are put together: using and training an auxiliary linear layer should improve the results by adding slightly more complexity to this integration.

As a last note, we would like to suggest continuing on the long-term prediction sub-task. Two future works on this could be to introduce waypoints as in [6] and to experiment on

different versions of the model: this task, in fact, has been performed when only the cosine function was implemented, thus it would be interesting to test it with the other improvements as well.

References

[1] T. Gu, G. Chen, J. Li, C. Lin, Y. Rao, J. Zhou, and J. Lu, "Stochastic trajectory prediction via motion indeterminacy diffusion," 2022. [Online]. Available: https://arxiv.org/abs/2203.13777

[2] J. Sohl-Dickstein, E. A. Weiss, N. Maheswaranathan, and S. Ganguli, "Deep unsupervised learning using nonequilibrium thermodynamics," 2015. [Online]. Available: https://arxiv.org/abs/1503.03585

[3] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," 2020. [Online]. Available: https://arxiv.org/abs/2006.11239

[4] A. Nichol and P. Dhariwal, "Improved denoising diffusion probabilistic models," 2021. [Online]. Available: https://arxiv.org/abs/2102.09672

[5] L. F. Chiara, P. Coscia, S. Das, S. Calderara, R. Cucchiara, and L. Ballan, "Goal-driven self-attentive recurrent networks for trajectory prediction," 2022. [Online]. Available: https://arxiv.org/abs/2204.11561

[6] K. Mangalam, Y. An, H. Girase, and J. Malik, "From goals, waypoints & paths to long term human trajectory forecasting," 2020. [Online]. Available: https://arxiv.org/abs/2012.01526

[7] D. Helbing and P. Molnár, "Social force model for pedestrian dynamics," *Physical Review E*, vol. 51, no. 5, pp. 4282–4286, may 1995. [Online]. Available: https://doi.org/10.1103%2Fphysreve.51.4282

[8] A. Alahi, K. Goel, V. Ramanathan, A. Robicquet, L. Fei-Fei, and S. Savarese, "Social lstm: Human trajectory prediction in crowded spaces," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 961–971. [Online]. Available: https://cvgl.stanford.edu/papers/CVPR16_Social_LSTM.pdf

[9] B. Ivanovic and M. Pavone, "The trajectron: Probabilistic multi-agent trajectory modeling with dynamic spatiotemporal graphs," 2018. [Online]. Available: https://arxiv.org/abs/1810.05993

[10] T. Salzmann, B. Ivanovic, P. Chakravarty, and M. Pavone, "Trajectron++: Dynamically-feasible trajectory forecasting with heterogeneous data," 2020. [Online]. Available: https://arxiv.org/abs/2001.03093

[11] N. Daniel, A. Larey, E. Aknin, G. A. Osswald, J. M. Caldwell, M. Rochman, M. H. Collins, G.-Y. Yang, N. C. Arva, K. E. Capocelli, M. E. Rothenberg, and Y. Savir, "Pecnet: A deep multi-label segmentation network for eosinophilic esophagitis biopsy diagnostics," 2021. [Online]. Available: https://arxiv.org/abs/2103.02015

[12] P. Dendorfer, A. Ošep, and L. Leal-Taixé, "Goal-gan: Multimodal trajectory prediction based on goal position estimation," 2020. [Online]. Available: https://arxiv.org/abs/2010.01114

[13] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," 2015. [Online]. Available: https://arxiv.org/abs/1505.04597

[14] A. Gupta, J. Johnson, L. Fei-Fei, S. Savarese, and A. Alahi, "Social gan: Socially acceptable trajectories with generative adversarial networks," 2018. [Online]. Available: https://arxiv.org/abs/1803.10892

[15] S. Pellegrini, A. Ess, K. Schindler, and L. van Gool, "You'll never walk alone: Modeling social behavior for multi-target tracking," in *2009 IEEE 12th International Conference on Computer Vision*, 2009, pp. 261–268. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5459260

[16] A. Mohamed, D. Zhu, W. Vu, M. Elhoseiny, and C. Claudel, "Social-implicit: Rethinking trajectory prediction evaluation and the effectiveness of implicit maximum likelihood estimation," 2022. [Online]. Available: https://arxiv.org/abs/2203.03057

[17] "Scipy gaussian kernel density estimation function," https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.gaussian_kde.html, accessed: 2023-01-15.

[18] D. E. Rumelhart and J. L. McClelland, *Learning Internal Representations by Error Propagation*, 1987, pp. 318–362. [Online]. Available: http://www.cs.toronto.edu/~hinton/absps/pdp8.pdf

[19] M. I. Jordan, "Serial order: a parallel distributed processing approach. technical report, june 1985-march 1986," 5 1986. [Online]. Available: https://www.osti.gov/biblio/6910294

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

[21] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," 2014. [Online]. Available: https://arxiv.org/abs/1409.1259

[22] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning Internal Representations by Error Propagation*. Cambridge, MA, USA: MIT Press, 1986, p. 318–362.

[23] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2013. [Online]. Available: https://arxiv.org/abs/1312.6114

[24] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014. [Online]. Available: https://arxiv.org/abs/1409.0473

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[26] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6296535

[27] A. Krizhevsky, "Learning multiple layers of features from tiny images," pp. 32–33, 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[28] A. Lerner, Y. Chrysanthou, and D. Lischinski, "Crowds by example," *Computer Graphics Forum*, vol. 26, no. 3, pp. 655–664, 2007. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2007.01089.x

[29] L. Perez and J. Wang, "The effectiveness of data augmentation in image classification using deep learning," 2017. [Online]. Available: https://arxiv.org/abs/1712.04621

[30] A. Robicquet, A. Sadeghian, A. Alahi, and S. Savarese, "Learning social etiquette: Human trajectory understanding in crowded scenes," in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 549–565. [Online]. Available: https://svl.stanford.edu/assets/papers/ECCV16social.pdf

[31] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[32] "Trajectron++ bug in computing derivatives," https://github.com/StanfordASL/Trajectron-plus-plus/issues/40#issue-856642520, accessed: 2023-01-15.

[33] "Numpy gradient calculation," https://numpy.org/doc/stable/reference/generated/numpy.gradient.html, accessed: 2023-01-15.

[34] L. Biewald, "Experiment tracking with weights and biases," 2020, software available from wandb.com. [Online]. Available: https://www.wandb.com/

[35] E. Luhman and T. Luhman, "Knowledge distillation in iterative generative models for improved sampling speed," 2021. [Online]. Available: https://arxiv.org/abs/2101.02388

[36] P. Dhariwal and A. Nichol, "Diffusion models beat gans on image synthesis," 2021. [Online]. Available: https://arxiv.org/abs/2105.05233

[37] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, "Hierarchical text-conditional image generation with clip latents," 2022. [Online]. Available: https://arxiv.org/abs/2204.06125

[38] C. Saharia, W. Chan, S. Saxena, L. Li, J. Whang, E. Denton, S. K. S. Ghasemipour, B. K. Ayan, S. S. Mahdavi, R. G. Lopes, T. Salimans, J. Ho, D. J. Fleet, and M. Norouzi, "Photorealistic text-to-image diffusion models with deep language understanding," 2022. [Online]. Available: https://arxiv.org/abs/2205.11487

A

Variance learning plots

In this appendix we present the comparison plots between the model without the learned variance and the model with the learned variance.

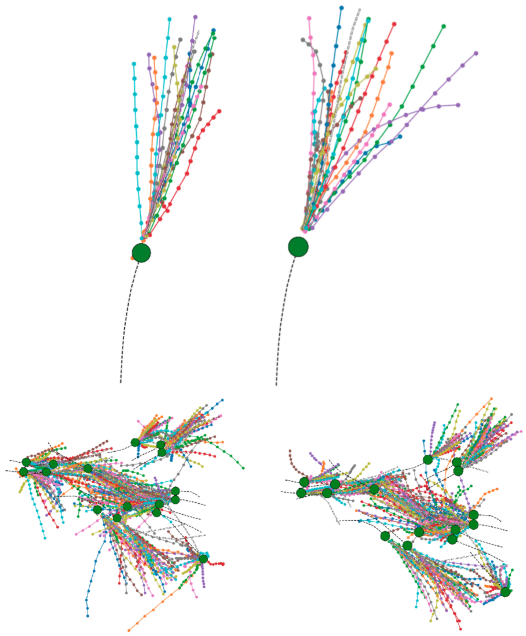

Figure A.1: Comparison between predictions on the Hotel and Univ datasets using the non learned version of the model (on the left) and the learned version (on the right). Note that, as the results in Chapter 6 suggest, learning the variance on the Hotel dataset worsen the likelihood.

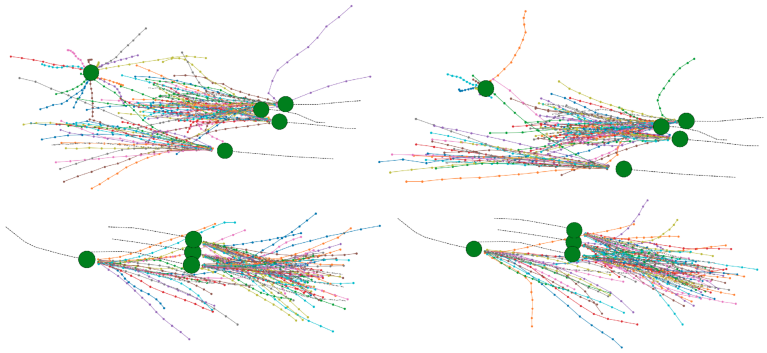

Figure A.2: Comparison between predictions on the Zara1 and Zara2 datasets using the non learned version of the model (on the left) and the learned version (on the right).

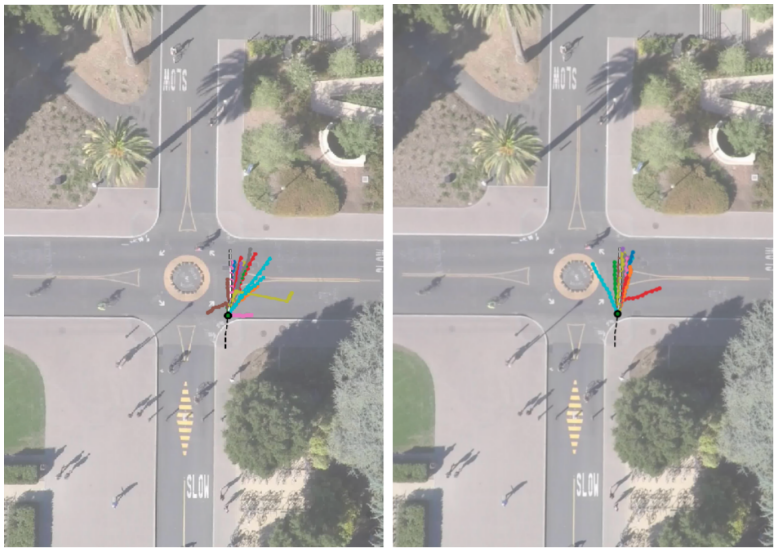

Figure A.3: Comparison between predictions on the SDD dataset using the non learned version of the model (on the left) and the learned version (on the right).

Acknowledgments

I would like to thank my supervisor Lamberto Ballan and my co-supervisor Sourav Das for guiding me on this path and leaving me to experiment with my knowledge; their assistance and their suggestions have been paramount many times during this thesis work. I would also like to thank the University of Padova, which allowed me to learn a lot of beautiful stuff during these years, and the University of Helsinki, which hosted me for one year making me feel at home and giving me inspiration for my future life.

Of course, I would like to give my best gratitude to my family and all of my friends for all the support they gave me through these years. This might sound prosy, but I truly believe I would not be here if it wasn't for you guys: I can't name every one of you, but you know you are always in my thoughts.