POLITECNICO DI TORINO

Master's Degree in Data Science and Engineering

Master's Degree Thesis

Transformer based stochastic approach to pedestrian trajectory forecasting

Supervisors

Prof. Lia MORRA

Dott. Simone LUETTO

Candidate

Giacomo GARRONE

April 2022

A mio papà,
questa è di entrambi

Abstract

Pedestrian trajectory forecasting is an active field of research in which the goal is to predict the path of an agent over a period of time, given some previously observed past motion history. This predicted period of time can be long or short, leading to long-term forecasting or short-term forecasting. In this thesis, we deal with short-term trajectory forecasting which consists in a forecasting window of about five seconds in the future. Having a model capable of these kinds of predictions is useful in many applications such as for service robots, autonomous driving or anomaly detection, where predicting a possible outcome is necessary to plan future moves.

A great challenge in modelling human motion behaviour is the fact that it is not deterministic by nature. An agent movement and trajectory is determined by many forces but also by the own will of the agent, which is not predictable with absolute certainty. In fact, multiple future trajectories may be plausible in certain situations such as in front of a fork in the road, or to avoid a collision, and even if the intent and destination of the agent are known, there are possibly infinite trajectories which can be possible. This stochastic and multimodal behaviour is the main focus of this thesis. With the advent of the Transformer model for Natural Language Processing, more and more Transformer based models have been proposed for sequence modelling problems. Motivated by their good performance with respect to Recurrent Neural Networks also for this task, the experiments of this thesis have been conducted using Transformer based architectures. In the first part of the experiments, the used model from literature, Transformer TF, is introduced and different preliminary experiments are carried out on the model to improve its performances, such as lowering model size, changing the inputs and perform a data augmentation technique which is proven to be beneficial. Then, starting from the previous deterministic model, a proposal is made to take into account the problem stochasticity. The proposed model, Transformer GAN, is a generative model capable of multiple predictions thanks to a sampled random latent vector. To efficiently generate multiple and multimodal predictions, a sampling refinement technique is then introduced which drastically improves the performances of the stochastic approach used. The results of the Transformer GAN after the sampling refinement method reach state-of-the-art performances with respect to other deep learning models for trajectory forecasting that do not use any social nor map information.

Acknowledgements

First of all, I would like to thank my advisor, Prof.ssa Lia Morra, for her availability, help and tutoring for this thesis.

A big thank you also to the people working at Addfor for the possibility to work with them and for sharing their resources with me, making this work possible. In particular, I would like to thank Simone and Rosalia that accompanied me through all these months, giving great advice and support.

Then, I would like to thank all the friends that accompanied me in these years. I feel very lucky to be surrounded by incredible people. Thank you to all the friends of Lavori in Corso, your daily presence made the Poli feel like a second home. I would also want to thank the CLU, where I received much more than I deserved. Thank you to all the people for being a part of this wonderful journey.

How to forget then the Breakfast Club, Anna, Ele, Fra and Richi. Sharing countless breakfasts and Quattro Soldi lunches with you cheered up even the worst days at Poli.

Thank you to my childhood friends that have been constantly part of my journey growing up, Abel, Massa, Enri, Aime, Silva. I could not have asked for better friends and, even if we don't see each other every day like it used to be, every time it seems like never changed.

Thank you to Edo, a truly great friend. I cannot be grateful enough for your help through all these years. Thank you for always caring about everyone and never wanting anything in return.

Thank you to my family, this surely would not have been possible without your help and care. Despite my terrible character you have always been present supporting me at every step of the way. To my father Giulio, my mother Cecilia and my brothers Gabriele and Giovanni, for their testimony and constant reminder

of what is really important.

And lastly thank you to Francesca. I would not even know where to start in order to thank you properly. Thank you for your kindness, patience and support shown daily and in every situation. Your presence is the greatest gift of my life.

Table of Contents

List of Tables VII

List of Figures VIII

1 Introduction 1

2 Trajectory forecasting 3
- 2.1 Problem formulation . . . 3
  - 2.1.1 Regression and classification . . . 4
  - 2.1.2 Deterministic and stochastic . . . 5
- 2.2 Applications . . . 6
- 2.3 Objectives and challenges . . . 6
- 2.4 Pre-deep learning . . . 7
  - 2.4.1 Kinematic models . . . 7
  - 2.4.2 Social force model . . . 8
  - 2.4.3 Energy minimization model . . . 9

3 Introduction to deep learning 11
- 3.1 Introduction . . . 11
  - 3.1.1 Neural networks . . . 12
  - 3.1.2 Backpropagation . . . 14
  - 3.1.3 Optimizers . . . 17
- 3.2 Recurrent neural networks . . . 20
  - 3.2.1 LSTM . . . 22
- 3.3 Attention . . . 24
  - 3.3.1 Global or local attention . . . 25
  - 3.3.2 Hard attention . . . 26
  - 3.3.3 Soft attention . . . 26
  - 3.3.4 Self attention . . . 28
- 3.4 Transformers . . . 28
  - 3.4.1 Transformer architecture . . . 29

3.4.2 Attention mechanism . . . 29
3.4.3 Encoder . . . 33
3.4.4 Decoder . . . 34
3.4.5 Input embedding . . . 34
3.4.6 Positional encoding . . . 35
3.5 Generative models . . . 37
3.5.1 GANs . . . 37
3.5.2 CGANs . . . 37
3.5.3 Variational Autoencoders . . . 38
3.5.4 Conditional Variational Autoencoders . . . 41

4 Deep learning models for trajectory forecasting 43
4.1 RNN-based models . . . 44
4.1.1 Social LSTM . . . 44
4.1.2 Social GAN . . . 46
4.1.3 Trajectron++ . . . 47
4.2 Transformer based models . . . 48
4.2.1 Transformer TF . . . 48
4.2.2 AgentFormer . . . 49

5 Experiments 52
5.1 Introduction . . . 52
5.2 Datasets . . . 52
5.2.1 ETH and UCY . . . 53
5.3 Metrics . . . 54
5.3.1 Average Displacement Error (ADE) . . . 54
5.3.2 Final Displacement Error (FDE) . . . 55
5.4 Chosen model . . . 55
5.4.1 Input embeddings dimension . . . 56
5.4.2 Loss . . . 60
5.4.3 Coordinates encoding . . . 61
5.4.4 Data augmentation . . . 64
5.5 Stochastic model . . . 66
5.5.1 Proposal: Transformer GAN . . . 67
5.5.2 Sampling method . . . 71
5.5.3 Results . . . 77
5.5.4 Predictions comparison before and after sampling method . 78
5.5.5 Literature comparison . . . 80

6 Conclusions 82

Bibliography 84

List of Tables

5.1 Results of the original Transformer TF following the single trajectory deterministic protocol. Results are reported from original paper [7]. The models are trained on 4 datasets and tested on the remaining one, following the Leave-One-Out approach. 57
5.2 Number of learnable parameters in Transformer TF while varying the embedding dimension. By decreasing embedding size we speed up the training and have a less complex model, while keeping the performances similar for this task. 58
5.3 ADE/FDE metrics on ETH-UCY datasets while varying the embedding dimension. 512 originally used in TF. 60
5.4 ADE/FDE results using a 2-norm loss and a 1-norm loss. 2-norm originally used in TF. 61
5.5 Results of the models using the different coordinates encodings. Relative encoding is the one originally used in TF. 64
5.6 Results of the experiments on data augmentation techniques. 66
5.7 Overall results of Transformer GAN prior and after the sampling method. The improved deterministic Transformer TF results are also shown for completeness. The number before S refers to the number of samples predicted. The metrics used for the stochastic models are the TopK-ADE and TopK-FDE with K equal to the number of samples drawn. 77
5.8 Comparison between literature results of stochastic models using a TopK-ADE and TopK-FDE with K=20 samples. The results are taken from publications. Our model is the Transformer GAN generator after the sampling method. The first three models are the only comparable in terms of inputs as they consider only the trajectory information. The others use also map or social information 81

List of Figures

2.1 In white the coordinates at time steps $t = 1, ...T_{obs}$, in red and cyan the predicted and ground truth coordinates respectively at $t = T_{obs+1}, ..., T_{pred}$. Image from [10] 4

2.2 Classification and regression. On the left an example of discretization of the input scene that leads to a grid based approach with classification. On the right a non grid based approach that represents the inputs and outputs as real values as a regression task. Image from [9]. 5

2.3 Social force model with attractive and repulsive forces that govern the motion. Image from [11] 8

3.1 Perceptron (from [17]) 12

3.2 Activation functions (from [18]) 13

3.3 MLP structure (from [19]) 14

3.4 Momentum representation (from [15]). Black arrows indicate the step that gradient descent would take in that point, while the red line indicates the path followed with momentum. 19

3.5 On the left a general RNN structure. On the right the equivalent unfolded version where every time step is represented with the same repeated structure. x_t and h_t are the input token and relative hidden state at time step t respectively. In the figure, the passage from h_t to y_t it's not represented (from [22]). 22

3.6 RNN cell architecture (left) vs LSTM cell architecture (right). From [22]. 23

3.7 Global and local attentions from [29]. Each blue box is an input token hidden state, while the grey boxes are the target tokens states after transformation from attention. We can observe that the context vector c_t is computed with different source tokens between global and local attention (not the same connections to the context vector). 25

3.8 Example of soft attention on a English to French translation task. The brightness of the squares indicates the attention weight between the two words during the generation phase. It is interesting to notice that the model learned to align correctly the adjectives which are used in different orders from English to French (from [28]). 27
3.9 Soft vs hard attention. Soft attention is a global attention differentiable over the entire input domain, while hard attention focuses only on a state of the input sequence and it is a local attention to only one state, it typically requires reinforcement learning techniques to be trained. 28
3.10 Transformer architecture as proposed in [6]. The grey block on the left is the encoder block, while on the right is the decoder block. . . 30
3.11 Scaled dot-product attention operations (from [6]) 31
3.12 Multi-head attention as in [6] 33
3.13 First 50 positional encodings using a d_{model}-dimensional vector of 128. Each row represents a positional encoding vector. From [33]. . 36
3.14 Conditional GAN architecture as proposed in the original paper [35]. Both the discriminator and generator are additionally fed the y conditioning (in green). During training, alongside the y conditioning, the input to the discriminator can be either the generator output or the sample from the training set (light grey lines). 38
3.15 VAE architecture from [37]. Encoder and decoder are trained to maximize the ELBO. The encoder learns the mapping from x to z, thus $q(z|x)$, while the decoder learns the mapping from z to x, $p(x|z)$. 41
3.16 Latent variable z distribution in the case of a VAE (left) or CVAE (right). This example is from a model trained on MNIST dataset with a 2D latent vector dimension. Each color corresponds to a digit between 0 and 9. We can observe how in VAE latent space the different digits occupy different portions of the space and similar digits are close to each other. In the case of CVAE instead, we do not have any distinction between the digits in the overall latent space, but if we consider each digit distinctly, every distribution $q(z|x, c = c)$ follows a Normal distribution $\mathcal{N}(0,1)$. 42

4.1 Standard data-driven pipeline for pedestrian trajectory forecasting. From [9]. 44
4.2 Social LSTM model from [3]. Separate LSTMs model each one of the agents and are interconnected by the social pooling layer which allows neighbor information to be encoded. In the bottom a detail of the social pooling layer for one person. 45
4.3 Social GAN architecture from [4]. 46

4.4 Trajectron++ architecture from [5]. In the gray box, the modelling of agent history at the top, the modelling of interactions in the middle and the inclusion of map information at the bottom. In orange the encoding of future steps used in training for the CVAE. . 47
4.5 Transformer TF architecture, from [7]. 49
4.6 AgentFormer architecture, from [8]. 50

5.1 Frames of Zara scene (left) and Univ scene (right). Image coordinates are extracted from video and transformed to world coordinates. . . 54
5.2 ADE metric in the top, the distance between true and predicted position is averaged across timesteps. FDE metric in the bottom, only the last timestep is used to compute the distance. 56
5.3 Training curves for the Zara2 dataset with diverse embedding dimensions. On the top the training loss, on the bottom the ADE and FDE, left and right respectively, computed on the validation set. Orange is 512 emb., green is 256 emb., light blue 128 emb. and blue is 64 emb. 58
5.4 Learning rate scheduler for different embedding dimensions. The warmup steps are 3000. 59
5.5 Representation of different coordinates encodings. On the top raw coordinates, bottom left coordinates centered in the first timestep, bottom right coordinates centered on the last observed timestep. . . 63
5.6 Training curve of a model tested on the Hotel dataset. On the left the ADE and on the right the FDE. We can observe the performance differences between raw coordinates (blue), coordinates centered in the first point (light blue), coordinates centered in T_{obs-1} (red) and relative coordinates (magenta). 63
5.7 Example of original trajectory (left) and rotated trajectory (right) . 65
5.8 Transformer GAN architecture. 68
5.9 Learning curves for the GAN. ETH magenta, Hotel grey, UCY blue, Zara1 green and Zara2 orange. 70
5.10 DLow architecture applied to human motion forecasting. The network Q outputs the parameters of the transformations that are applied to ϵ to obtain the k latent codes z_i. The multiple latent codes are fed to the generator to produce multiple and diverse outputs. From [42]. 73
5.11 Proposed sampling method architecture, DLow with fine-tuning of the generator parameters. The Q network outputs the parameters that are used to transform the sampled vector ϵ in k latent vectors that generate k predictions. 74

5.12 Learning curves for the sampling method. ETH brown, Hotel magenta, UCY grey, Zara1 light blue and Zara2 green. 76
5.13 Comparison between TF GAN prior (top) and after (bottom) sampling method on an example from Hotel dataset. On the left 20 predicted trajectories, on the right the best one. 78
5.14 Comparison between TF GAN prior (top) and after (bottom) sampling method on an example from UCY dataset. On the left 20 predicted trajectories, on the right the best one. 79
5.15 Comparison between TF GAN prior (top) and after (bottom) sampling method on an example from Zara1 dataset. On the left 20 predicted trajectories, on the right the best one. 80

Chapter 1

Introduction

This thesis investigates the task of trajectory forecasting, more specifically pedestrian trajectory forecasting, by using deep learning models and methodologies. The task of pedestrian trajectory forecasting consists in predicting the future motion of a pedestrian, given some past observations. There are two possible types of trajectory forecasting, long term or short term based on how much in the future we want to forecast. In this thesis we specifically focus on the task of short term trajectory forecasting by predicting the motions for about five seconds.
Various applications can benefit from this work such as service robots which operate in environments shared with humans and that should be able to anticipate human motion to plan their actions, self driving vehicles in which the prediction of future motion is also critical for safety reasons, or surveillance cameras which can detect anomalies in an agent motion if it moves drastically different from its predicted behaviour.
Modelling human motion is not an easy task and presents many challenges. It consists in two main dimensions to model, the temporal sequence dimension and the social interactions dimension, with the latter that must be able to handle a variable number of agents in a scene. Also, the proposed trajectories should be physically and socially acceptable and lastly, human motion is not deterministic which means that a solution is not unique for this type of task and multiple futures are plausible.
This work focuses the most on the stochastic aspect of the problem, by proposing a generative model capable of multiple predictions and a refinement step to improve its performance on the stochastic pedestrian trajectory forecasting task. One of the main difficulties for a stochastic model is the capability to learn a multimodal distribution for the trajectories and the proposed architecture tries to model this aspect.
Many models have been developed in the years for this task, from physics-based models such as the constant velocity model [1] or the very influential Social Force

model [2], up to deep learning models based on Recurrent Neural Networks such as Social LSTM [3], followed by Social GAN [4] and Trajectron++ [5] to name a few. Some of these models are capable of multiple predictions, some of them instead focuses more on modelling social interactions or both. With the advent of the Transformer architecture [6], new models were introduced such as Transformer TF [7] or AgentFormer [8] which is currently the state-of-the-art model alongside Trajectron++ for the stochastic approach.
From the introduction of the Transformer, different works have demonstrated its performances in sequence modelling tasks, such as in natural language processing problems. For this reason, in this work a Transformer based approach to the trajectory forecasting task is investigated. The initial model used for the experiments is Transformer TF for its trade-off between simplicity and performance. A first part of the experiments is dedicated to investigating the model performances under some hyperparameters change, while a second part is dedicated to modelling the stochasticity of human motion. In this second part a novel model is proposed and tested in different settings and finally a sampling technique is proposed to improve its stochastic performances by effectively modelling the multimodality. All the experiments have been performed in collaboration with Addfor Industriale S.r.l.

The thesis is structured as follows. In Chapter 2 the problem of trajectory forecasting is described, with its application, challenges and models used before the advent of deep learning. In Chapter 3, an introduction to deep learning is presented, the main components and the description of important architectures such as Recurrent Neural Networks and the Transformer. In Chapter 4 a literature review is done and some of the most important and current state-of-the-art models for the trajectory forecasting task are described. In Chapter 5 the experiments and the results are discussed, the datasets and metrics used are introduced, experiments are conducted on the chosen model from literature and then a model proposal is done to tackle the stochastic and multimodal behaviour of the task. Finally, a literature comparison is presented to compare the results obtained with the proposed architecture. In Chapter 6 the conclusions are drawn.

Chapter 2

Trajectory forecasting

2.1 Problem formulation

Pedestrian trajectory forecasting consists in forecasting the movement of the people which conforms to the social norms. To formally define pedestrian trajectory forecasting, firstly we have to introduce the notion of *Trajectory* and *Scene*.

- *Trajectory* is defined as the time-profile of pedestrian motion states, which are generally the position coordinates sampled every t seconds. Additional information can be used too (e.g. body pose, gaze direction) to add value to each agent motion states.
- A *Scene* is a collection of trajectories of multiple humans interacting in a social setting. A scene could also comprise physical objects and non-navigable areas that affect the pedestrian trajectories.

With these notions it is then possible to define pedestrian trajectory forecasting as:

Given the past trajectories of all pedestrian in a scene, forecast the future trajectories which conform to the social norms. [9]

Trajectory forecasting is primarily a sequence modelling task. The time t is discretized and the positions of every pedestrian are observed at time steps $t = 1, ...T_{obs}$. Finally, the future positions of each agent are then forecasted from time steps $t = T_{obs+1}$ to $t = T_{pred}$.

The predicted trajectories can also be more than one in a multimodal setting, as it is observable in Figure 2.1. To tackle the task, firstly it is needed for the spatial coordinates of the agents to be extracted from videos or sensors using a functioning system that allows for detection and tracking of the agents. After this step, it is possible to use them as inputs to the model. In this thesis the focus is

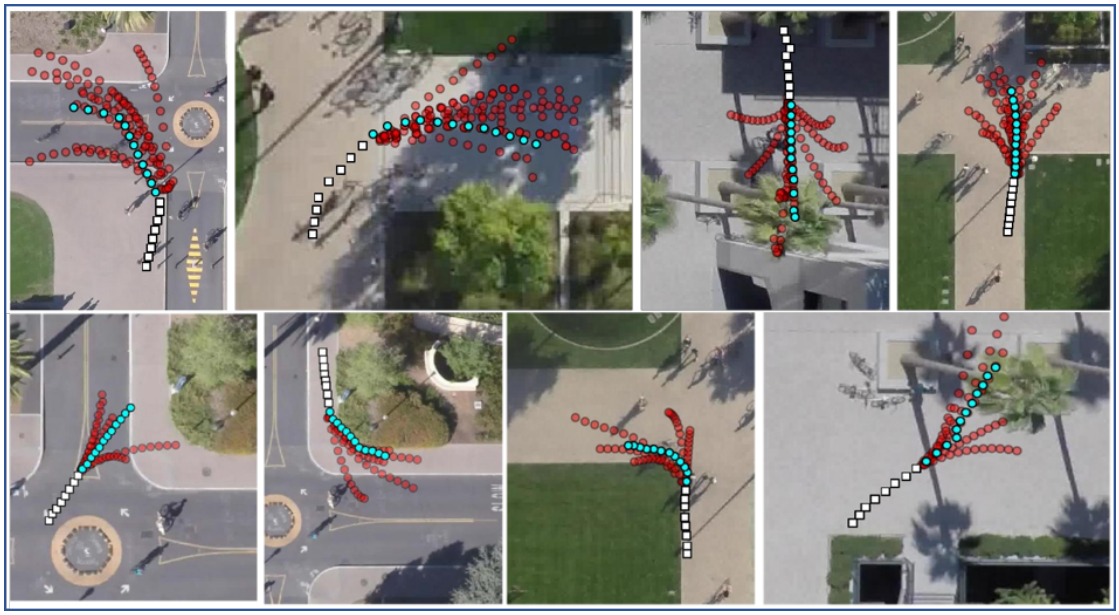


Figure 2.1: In white the coordinates at time steps $t = 1, ...T_{obs}$, in red and cyan the predicted and ground truth coordinates respectively at $t = T_{obs+1}, ..., T_{pred}$. Image from [10]

on the trajectory prediction module and not on the entire pipeline, considering the inputs from the detection and tracking system as already given.

2.1.1 Regression and classification

The trajectory forecasting task can be tackled using two different approaches: a regression or a classification approach. This depends on how the inputs and the outputs are codified. Based on this distinction it is possible to categorize the models in two: grid based and non-grid based.
Grid based models are models which tackle the task using the classification approach. In these models, the scene is represented as a grid where each cell is a possible position in the scene. The input to the model is the whole grid at each timestep and the output trajectory is a sequence of cell representing the future occupied cells for the considered agent. It is a classification task because the input scene is discretized and the model at each step chooses in which cell the agent is going to move. This type of approach is mainly used for its simplicity of representing the input scene, which is of fixed length. Furthermore, each cell can also contain other information than only the agent which is occupying it, such as his velocity, direction etc. The main drawback of this approach is the approximation error obtained by discretizing the scene both in input and output. Obviously, the more

grid cells is subdivided the scene, the less the less the approximation error going to be, but using more grid cells makes the model more complex.
The other approach is the regression one used by the non-grid based models. In these models, the inputs are agent states, which usually are the agent coordinates (x, y) with some optional additional information, and the outputs are the agent coordinates codified in real values. That is the reason why it is a regression problem. Most of the state-of-the-art models use this approach which usually gives better results than the grid based approach.

A representation of the two approaches is visible in Figure 2.2.

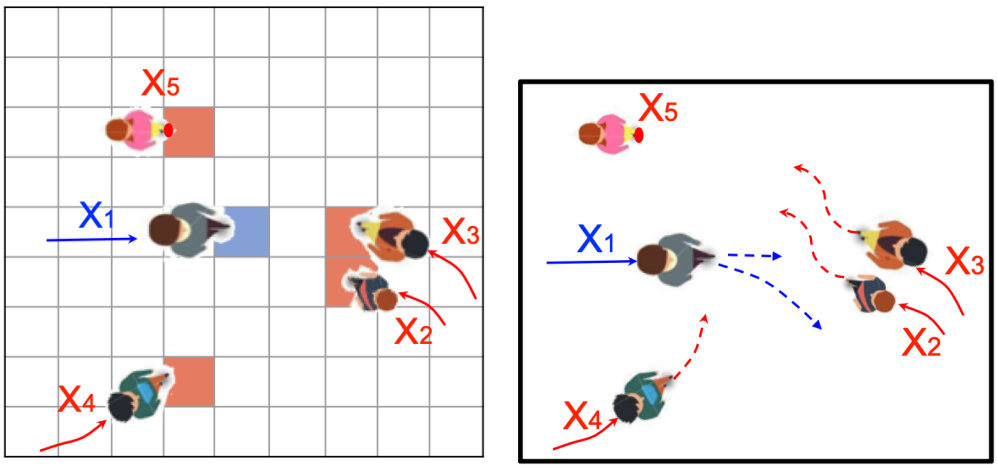


Figure 2.2: Classification and regression. On the left an example of discretization of the input scene that leads to a grid based approach with classification. On the right a non grid based approach that represents the inputs and outputs as real values as a regression task. Image from [9].

2.1.2 Deterministic and stochastic

Another categorization of the models for trajectory forecasting is the distinction between deterministic and stochastic models.
Deterministic models are models that from one observed scene are capable of a single prediction, which is the most likely among the multiple possible ones. This approach does not follow the real human motion behaviour which is not deterministic. In fact, multiple trajectories are possible for an agent, even to reach the same destination, and most of the times there is not a correct or wrong trajectory to choose.
Stochastic models follows this intuition and are capable of multiple predictions from the same past observed trajectory. Usually these multiple futures predicted

try to cover minor changes in the trajectory (such as a little direction change) or a multimodal future in which the agent can move in various different directions (such as an agent in front of a fork in the road). Many different models were proposed in literature trying to model the stochasticity in human motion and some of them are described in the next chapters.
This thesis aims at focusing more on the stochastic approach to the trajectory forecasting task.

2.2 Applications

Pedestrian trajectory forecasting is a key task in different scenarios. Knowing and predicting the future movements of agents can be beneficial mainly in this domains [11]:

- *Service robots*: mobile service robots operate in environments that are shared with humans. Anticipating their motion can be crucial for a safe human-robot interaction and an efficient motion planning.

- *Self-driving vehicles*: an important field of application is autonomous driving, where the ability to accurately predict pedestrian and other road users future motion is critical for many reasons. It is different from the service robot domain since the velocities and masses are higher, and the potential harm for a bad forecasting module is much larger. Furthermore, vehicles need to operate in rapidly changing settings and have real time operating constraints.

- *Surveillance*: video surveillance of vehicles or human crowds can also be an application, since knowing a likely trajectory of the agents can be used to detect anomalies and unexpected behaviours.

2.3 Objectives and challenges

The task of trajectory forecasting poses multiple challenges that are addressed in diverse solutions. The main challenges are:

- *Sequence modelling*: it is necessary learn how to model short-term and long-term dependencies in the past trajectory.

- *Variable number of agents*: since we are dealing with a non-static scene and environment, we need to be capable of handling a variable number of agents entering and leaving the scene at different times.

- *Physically acceptable outputs*: a good pedestrian trajectory forecasting model should provide physically acceptable outputs, for example avoiding collisions. This is crucial in safety critical applications.

- *Social interactions*: the trajectory of an agent is affected by other agents in its surroundings. Modelling how an observation of a trajectory affects another trajectory is a difficult challenge and an active field of research.

- *Stochasticity*: the task is stochastic in the sense that given the history of a scene, there are multiple plausible futures (e.g. a pedestrian at an intersection), so it is not deterministic by nature. In that case, having no additional knowledge on the goal of the agents makes the acceptable possible outputs multiple. This aspect is very difficult to model since usually we have only one ground truth trajectory per agent, and it is not straight forward to evaluate the goodness and plausibility of an output.

All these challenges are extensively investigated in the literature using various methods. This work aims at tackling some of these challenges while focusing the most on the multimodality aspect, since it is one of the most difficult to model. The objectives of this project are to study and research techniques used to achieve this goal accurately, developing a novel model capable of multiple predictions that exploits these findings and compare the results with already available models from literature to find differences and improvements.

2.4 Pre-deep learning

In recent years, increasingly more deep learning based models are proposed to solve the task of pedestrian trajectory forecasting, but prior to the advent of deep learning, the majority of the works in this area were physics-based models. In these models, motion is predicted by defining an explicit dynamical model f based on Newton's law of motion. This section illustrates some important and influential physics-based works.

2.4.1 Kinematic models

Kinematic models are the simplest kind of models to use for trajectory forecasting. They do not consider any force that govern the motion. Most popular examples include the constant velocity model (CVM), constant acceleration model (CAM) [1] and coordinated turn model (CTM). The constant velocity model assumes piecewise constant velocity based on the last two observed frames of the agent. Denoting $\mathbf{p}_i^t$ the coordinates (x_i^t, y_i^t) of agent i at time step t, it computes the point

velocity as $\Delta_i = \mathbf{p}_i^t - \mathbf{p}_i^{t-1}$ and predicts the future positions as

$$\mathbf{p}_i^t = \mathbf{p}_i^{T_{obs}} + (t - T_{obs})\Delta_i \quad t \in T_{obs+1}, ..., T_{pred}$$

Despite its simplicity, the CVM performs relatively well and has competitive results. However, it has a lot of drawbacks as it assumes that the pedestrian will continue to walk with the same velocity and direction of the last two time steps. It is also very sensitive to measurement noise having no type of any filtering, and more importantly, it does not take into account neither the environment or other pedestrians. It also cannot predict curving trajectories.
Similarly to CVM, the constant acceleration model uses the last three time steps and propagates the same acceleration in the future time steps.
Lastly, the coordinated turn model assumes instead constant turn rate and speed. All these models are good approximations but do not model explicitly interactions and cannot predict complex behaviours as avoiding obstacles, people that are simple social norms.

2.4.2 Social force model

The social force model [2] is one of the most popular examples in the physics-based models. It was developed in 1995 for crowd analysis and the key idea is that the behaviour of a pedestrian, but also other agents, is guided by attractive and repulsive forces from other agents and obstacles (see Figure 2.3). It is still very influential since several works use the idea to improve the predictions. In the model,

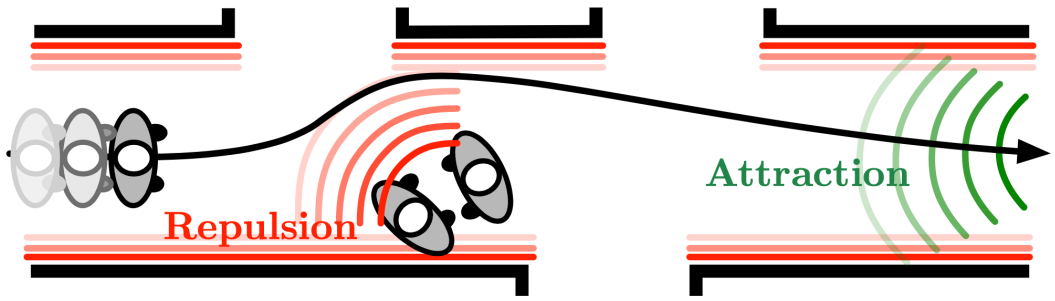


Figure 2.3: Social force model with attractive and repulsive forces that govern the motion. Image from [11]

the pedestrian behaviour is governed by three forces:

- A term describing the acceleration towards the desired velocity of motion.

- Terms reflecting that a pedestrian keeps a certain distance to other pedestrians and obstacles (sum of repulsive forces).
- A term which models attractive effects (sum of attractive forces).

The resulting temporal changes $\frac{d\mathbf{w}_\alpha}{dt}$ of the preferred velocity $\mathbf{w}_\alpha(t)$ of agent α are defined by

$$\frac{d\mathbf{w}_\alpha}{dt} := \mathbf{F}_\alpha(t) + fluctuations$$

Where $\mathbf{F}_\alpha(t)$ is the sum of all the aforementioned forces and $fluctuations$ term takes into account random variations of the behaviour.
This way of formulating the problem derived from the observation that crowd pedestrian behaviour has striking analogies with gases and fluids. The difference is that the forces are not explicitly excerted by the environment on the pedestrian body, but are rather a quantity that describes the motivation to act. A pedestrian sense and perceive the environment and acts as if it has a force on his/her motion.

2.4.3 Energy minimization model

Following the social force model, an important work is the one formulated by Yamaguchi et al [12]. They propose an agent-based energy minimization behavioural problem. Each pedestrian is viewed as a decision making agent that consider various factors to decide where to go next. This factors, which are personal, social and environmental, are used to build an energy function that is used by the model for an accurate behaviour prediction. Interestingly, this model estimates also the social relationship of the people (grouping) and the desired destination of the agents.
The energy function for each pedestrian expresses the desirability of possible directions of motion for the pedestrian. It is defined as

$$\begin{aligned} E_\theta(\mathbf{v}; s_i, \mathbf{s}_{-i}) = \lambda_0 E_{damping}(\mathbf{v}; s_i) + \lambda_1 E_{speed}(\mathbf{v}; s_i) \\ + \lambda_2 E_{direction}(\mathbf{v}; s_i) + \lambda_3 E_{attraction}(\mathbf{v}; s_i, \mathbf{s}_{A_i}) \\ + \lambda_4 E_{group}(\mathbf{v}; s_i, \mathbf{s}_{A_i}) + E_{collision}(\mathbf{v}; s_i, \mathbf{s_{-i}} | \sigma_d, \sigma_w, \beta) \end{aligned} \tag{2.1}$$

Where $\theta = \{\lambda_0, \lambda_1, \lambda_2, \lambda_3, \lambda_4, \sigma_d, \sigma_w, \beta\}$ are the parameters to be learnt, A_i is the social group of pedestrian i, s_i is the state variable of the agent, $\mathbf{s}_{A_i}$ is the set of state variables of the pedestrians in i's social group and $\mathbf{s}_{-i}$ the set of all state variables excluding i.
The six terms of the function are:

- *Damping*: penalizes sudden changes in the choice of velocity.
- *Speed*: it is assumed that pedestrians have a preferred speed, it penalizes a speed that deviates from the preferred speed.

- *Direction*: it indicates the correct direction to the assumed goal.
- *Attraction*: it models the fact that people in the same group tend to stay close to each other.
- *Grouping*: people in the same group tend to walk at similar speed and directions, it penalizes velocities different from the average of the group.
- *Collision*: agents try to avoid collisions with obstacles or other agents.

Having defined the energy function, the transition of position $\mathbf{p}$ and velocity $\mathbf{v}$ of pedestrian i from time t to $t + \Delta t$ follows

$$\mathbf{p}_i^{t+\Delta t} = \mathbf{p}_i^t + \mathbf{v}_i^{t+\Delta t}\Delta t$$

$$\mathbf{v}_i^{t+\Delta t} = \underset{\mathbf{v}}{argmin} E_\theta(\mathbf{v}; s_i^t, \mathbf{s}_{-i}^t)$$

And the θ parameters are optimized by fitting the energy function to the trajectories in the training set.
The approach was then improved by [13] by considering different parameters to encode different pedestrian behaviours (aggressive, mild, etc).

While the previously presented models reach quite accurate results in some scenarios, they all rely on precisely calibrated and hand crafted features, which require strong domain knowledge. With the rise of deep learning in recent years, these models were replaced and the approach shifted from physics-based models to data-driven ones.

Chapter 3

Introduction to deep learning

3.1 Introduction

Deep learning is a subset of the machine learning field which extracts patterns from data using neural networks (NN). Despite its current popularity and use, the field started developing in the 50's with the perceptron [14], a mathematical model of a neuron of the brain and the principal component of a type of modern neural network. Unfortunately, in such early years, the technology was too advanced for the times which led to the so called AI winter, the research lost interest in the subject. The reasons of the resurgence of interest in the topic should be attributed mainly to three reasons [15, 16]:

- *Big Data*: much larger datasets are available, they are easier to collect and to store. This is a huge advantage since neural networks require vast amounts of data in order to be trained.

- *Hardware*: previously available hardware was not sufficient to support training of deep learning models which requires high specs. In recent years, with the advancements in Graphics Processing Units (GPUs) and parallelization, it is more affordable.

- *Software*: the development of ready-to-use packages and libraries made the topic more attractive to research and experiment.

The term *deep* is used because neural networks are composed by a large sequence of stacked layers with many parameters to be tuned.

3.1.1 Neural networks

There are many kinds of neural networks, but to introduce the idea it is better to start from the simplest and first one: the **multilayer perceptron** (MLP), also called **deep feedforward network** or **feedforward neural network**. To understand the concept of MLPs, we first have to study their main building blocks, which are the perceptrons [14].

A representation of the perceptron can be seen in Figure 3.1. We can define:

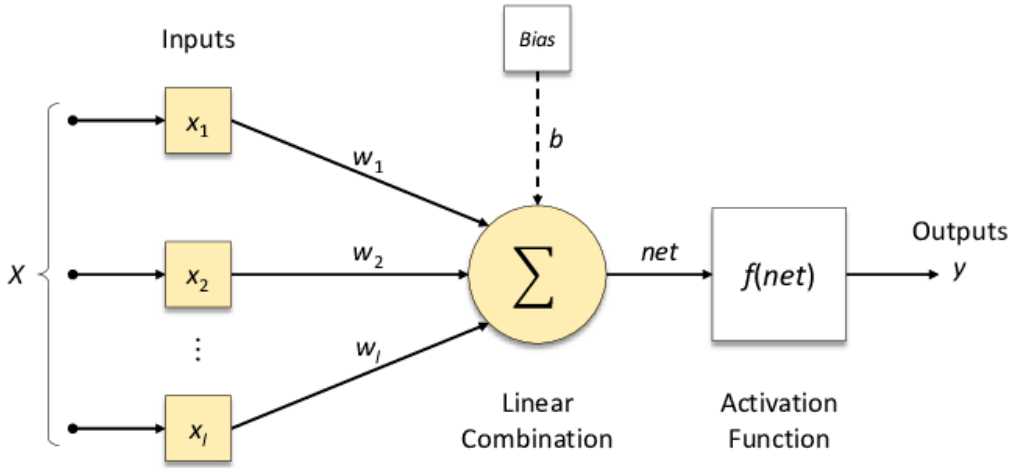


Figure 3.1: Perceptron (from [17])

- x_i the inputs.
- w_i the weights.
- b the bias term.
- $f(\cdot)$ as the non-linear activation function.
- y the output.

A mathematical representation of the output is given by:

$$y = f\left(b + \sum_{i=1}^{l} x_i w_i\right) \tag{3.1}$$

That is, the output of a single perceptron is simply a non-linear function applied to the sum of a bias term and the dot product between the vector of the inputs and the vector of the weights. The bias term only shifts the activation function and it is not influenced by the weights.

Why is it important to have a *non-linear* activation function? Because if we do not introduce a non-linearity in the model, we can only learn linear boundaries and cannot approximate complex functions, no matter how many perceptrons we stack on top of each other. In fact, a combination of linear functions will always result in another linear function. Activation functions are a really important concept in perceptrons and neural networks in general, there has been much research in the topic and now the most used one is called the *ReLU* (*Rectified Linear Unit*), given by:

$$f(x) = \max(0, x) \tag{3.2}$$

ReLU and its variants (*LeakyReLU, Parametric ReLU...*) are the most effective for neural networks because they are computationally lighter with respect to other functions since their derivative is close to the one of a linear function, and they help the models converge faster. The reason because we are dealing with derivatives is explained in Section 3.1.2.
Some of the most popular activation functions can be seen in Figure 3.2.

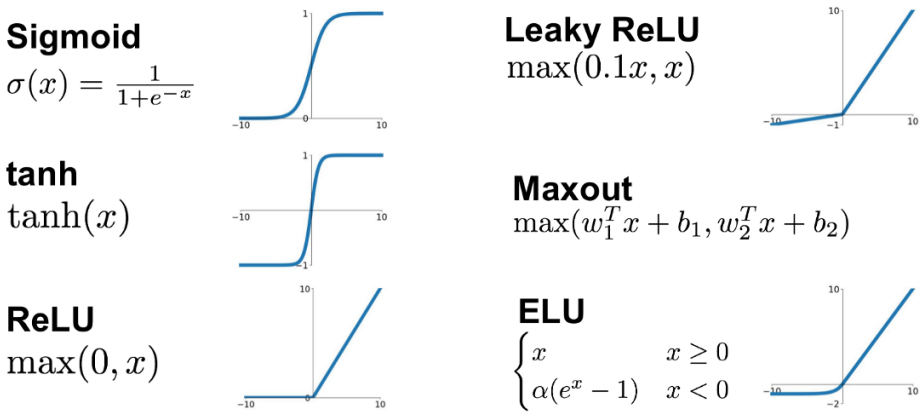


Figure 3.2: Activation functions (from [18])

Now we are ready to introduce the multilayer perceptron (Figure 3.3. It consists in many perceptrons used alltogether in various layers and stacked. There are three types of layers:

- *Input layer*: it is the first layer and it consists in the input nodes.
- *Hidden layers*: there is not a specific number of layers used, a neural network is *deep* because of the many stacked hidden layers. They process the input data and each layer is composed by many perceptrons, the number of nodes in each layer is called *width*.
- *Output layer*: it consists in one or more nodes depending on the task we are facing.

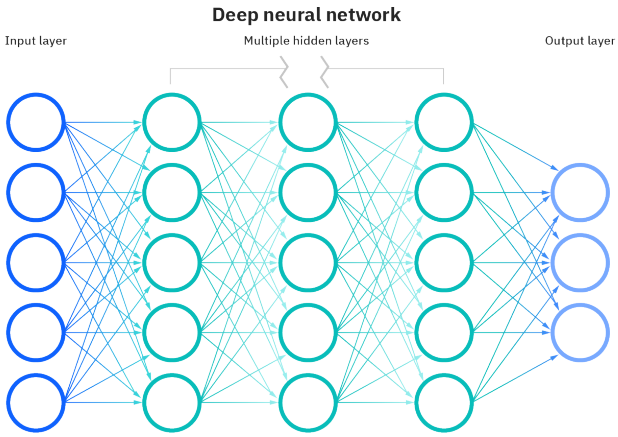


Figure 3.3: MLP structure (from [19])

Each node is fully connected with the previous layer. This type of model is also called *feedforward* because the inputs flow from the inputs to the outputs without feedback connections, as instead is a recurrent neural network (Section 3.2).

As stated before, this networks are represented by composing many different functions. Denoting y our output, we can write

$$y = f(\mathbf{x}) = f^3(f^2(f^1(\mathbf{x}))) \tag{3.3}$$

where f^1 is the *first layer*, f^2 the *second layer* and so on. During the training of the network, we want to find an f which is the closest to the actual f^*. This ability to closely approximate any function is what makes neural networks very powerful in many different conditions. One of the downsides of this generalization capabilities is that they easily have many parameters to learn. Any connection between nodes (weight) is one parameter to tune, so, it's easy to see that adding more nodes can drastically increase the number of parameters of the model.

3.1.2 Backpropagation

To train a neural network we need to introduce a measure, a distance from the true answer and the actual output of the model. This measure is typically called **loss**

$$\mathcal{L}(f(x^i; \mathbf{W}), y^i) \tag{3.4}$$

If the output of the model is not as expected, the value of the loss is going to be big indicating a poor performing model. When we have more than one sample in

the dataset, the total loss (called *empirical loss*) is averaged over all samples as

$$J(\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(f(x^i;\mathbf{W}), y^i) \tag{3.5}$$

This is also called *objective function* or *cost function*. There are many loss functions and the use of these depends on the task we are dealing with (e.g. classification or regression), the main takeaway is that, in any case, it measures a distance from the desired result.
We can observe that the objective function $J(\mathbf{W})$ is a function of the set of weights and biases $\mathbf{W}$ of the network. So, our aim is to minimize the value of this function by finding the best possible set of parameters $\mathbf{W}^*$

$$\mathbf{W}^* = \underset{\mathbf{W}}{argmin}\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(f(x^i;\mathbf{W}), y^i) = \underset{\mathbf{W}}{argmin}\ J(\mathbf{W}) \tag{3.6}$$

But how do we change the values of the weights to improve the loss? To do so, we have to know how the variation of each weight impacts the loss. This can be done using **backpropagation**.
Backpropagation is build upon the concept of the chain rule, which states that if we have $y = f(u)$ and $u = g(x)$ both differentiable, the derivative of y is

$$\frac{\partial u}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x} \tag{3.7}$$

That is, the derivative of a composite function can be split in sub-parts. Recalling 3.3, we can represent an MLP as a sequence of composite functions where we can apply the chain rule.
To explain backpropagation, we need to recap the notation used:

- $f(\cdot)$ is the activation non-linear function (ReLU, sigmoid, Figure 3.2)
- z_i^l is the value of neuron i in layer l before passing through f
- a_i^l is the activation value of neuron i in layer l
- w_{jk}^l is the weight between nodes j and k from layer l to $l+1$
- b_j^l is the bias of node j in layer l
- y is the output of the network
- $J(\mathbf{W})$ is the cost function

Considering an MLP with only 3 layers (1 input layer, 1 hidden layer and 1 output layer) and with the output layer consisting only in one neuron, we can simplify notation by denoting $y = a^3$, the output of the model is the activation of the third layer neuron. We can now write the partial derivative of the cost function $J(\mathbf{W})$ with respect to the weights between hidden and output layer as:

$$\frac{\partial J(\mathbf{W})}{\partial w_{j1}^2} = \frac{\partial J(\mathbf{W})}{\partial a^3}\frac{\partial a^3}{\partial z^3}\frac{\partial z^3}{\partial w_{j1}^2} \tag{3.8}$$

where we applied the chain rule going backward through the network. If we want to go back on the parameters of the first layer we need to recursively apply the chain rule and expand the previous equation as:

$$\frac{\partial J(\mathbf{W})}{\partial w_{jk}^1} = \frac{\partial J(\mathbf{W})}{\partial a^3}\frac{\partial a^3}{\partial z^3}\frac{\partial z^3}{\partial a_k^2}\frac{\partial a_k^2}{\partial z_k^2}\frac{\partial z_k^2}{\partial w_{jk}^1} \tag{3.9}$$

We can now see how the derivative of the activation function $\frac{\partial a_k^l}{\partial z_k^l}$ plays a role in the backpropagation algorithm.

We denote the first two terms of the previous equations as:

$$\delta^3 = \frac{\partial J(\mathbf{W})}{\partial a^3}\frac{\partial a^3}{\partial z^3} = \frac{\partial J(\mathbf{W})}{\partial a^3}f'(a^3) \tag{3.10}$$

as the *error* term which we send back from the output through the network. The third term of 3.8 is instead

$$\frac{\partial z^3}{\partial w_{j1}^2} = a_j^2 \tag{3.11}$$

and we can rewrite 3.8 as:

$$\frac{\partial J(\mathbf{W})}{\partial w_{j1}^2} = \delta^3 a_j^2 \tag{3.12}$$

Now, going one layer back, we see that:

$$\frac{\partial z^3}{\partial a_k^2} = w_{k1}^2 \tag{3.13}$$

and we rewrite 3.9:

$$\frac{\partial J(\mathbf{W})}{\partial w_{jk}^1} = \delta^3 w_{k1}^2 f'(a^2) a_j^1 \tag{3.14}$$

Generalizing, we note that the error δ^l in the previous layer is the weighted sum of the errors in the next layer δ^{l+1}, so we define:

$$\delta_j^l = f'(a^l)\sum_{i=1}^{m}\delta_i^{l+1}w_{ji}^l \tag{3.15}$$

So 3.14 becomes:

$$\frac{\partial J(\mathbf{W})}{\partial w_{jk}^1} = \delta_k^2 a_j^1 \tag{3.16}$$

And in general, for each neuron in every layer:

$$\frac{\partial J(\mathbf{W})}{\partial w_{jk}^l} = \delta_k^{l+1} a_j^l \tag{3.17}$$

$$\frac{\partial J(\mathbf{W})}{\partial b_j^l} = \delta_j^l \tag{3.18}$$

It is called backpropagation in fact, because the error in the output backpropagates through the network going backwards. In this way we can compute how each weight influences the output and change the values of the parameters accordingly, using optimizers which use this gradient information.

3.1.3 Optimizers

We have seen how to compute the gradients with respect to every parameter of the network. Once we have the gradients (which tell us the direction of major increase of the cost function) what do we do with such information?
We iteratively update the network parameters using an algorithm called optimizer, because it updates the network parameters trying to optimize and minimize the loss function. Almost the entirety of the optimizers are based on the *gradient descent algorithm* (see 1) where

- $\mathbf{W}$ are the network parameters
- $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ is the gradient of the loss function with respect to the network parameters
- η is a hyperparameter called *learning rate*

Algorithm 1 Gradient descent algorithm

Initialize parameters randomly $\sim \mathcal{N}(0, \sigma^2)$
while not converged **do**
  Compute gradient $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
  Update parameters $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
end while
return W

The gradient descent algorithm exploits the gradient direction information and updates the parameters in the opposite direction (the minus sign in the update

formula) which is the fastest decrease of the cost function.

The η hyperparameter is one of the most important in training. It represents how much of a step we want to take in that direction. It is fundamental for a proper training to set it right, since a not tuned learning rate can cause the loss function to bounce between values and diverge by never reach a local minimum, or to move too slowly to a solution remaining stuck to a false one or making the training very long.

There are three variants of the algorithm that differ in how much data we use to compute the gradient of the cost function [20]. By choosing the amount of data which takes to update the parameters, we make a trade-off between the accuracy of the parameter update and the time to make that update.

- *Batch gradient descent*: compute the gradient with respect to all training data. Very slow and cannot be used in online training or with large datasets.
- *Stochastic gradient descent (SGD)*: computes and performs an update for each training sample. Fast but the updates have an high variance which causes the cost function to fluctuate heavily. This high variance in updates enables SGD to jump to potentially better local minima, but at the same time makes the convergence much more difficult to the point that it is not guaranteed to converge.
- *Mini-batch gradient descent*: it is in the middle of the previous two and compute an update after n samples. Using this mini-batch decreases the variance of the parameters update leading to a more stable convergence and a faster training.

Since we are dealing with non-convex cost functions, finding the global minimum is a very difficult problem. With the gradient descent algorithm we can find a local minimum, but we do not know the goodness of the solution. This method is also highly affected by the learning rate, which is static and fixed for the whole training. To overcome these limitations, many other optimizers where invented that make use of an *adaptive learning rate* or some terms that benefit the optimization procedure.

Momentum

Gradient descent has some difficulties when dealing with areas where the cost function surface curves much more steeply in one direction than in another, where instead there is the local minimum. In these cases, the gradient oscillates between the slopes and approaches the local minimum very slowly, making the training too slow. Momentum is a method that helps accelerate gradient descent in the right

direction while minimizing these oscillations. It can be seen graphically in Figure 3.4. It does this by adding a term γ that adds a fraction of the past update vector

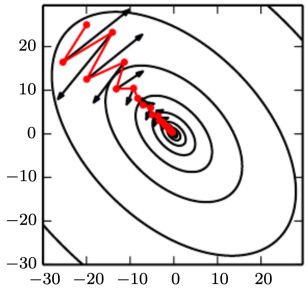


Figure 3.4: Momentum representation (from [15]). Black arrows indicate the step that gradient descent would take in that point, while the red line indicates the path followed with momentum.

to the current update vector:

$$v_t = \gamma v_{t-1} + \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} \tag{3.19}$$

$$\mathbf{W} \leftarrow \mathbf{W} - v_t \tag{3.20}$$

The result is similar to pushing a ball down a hill. It accumulates momentum as it gains velocity and the same happens to the network parameters. This makes the update of the parameters that change directions smaller, while accumulating for parameters that keep the same decrease direction, thus reducing the oscillation effect.

Adam

Adam (Adaptive Moment Estimation) is a method that computes adaptive learning rates for each one of the parameters. It uses an exponentially decaying average of past squared gradients v_t and an exponentially decaying average of past gradients m_t. The decaying averages are computed as:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} \tag{3.21}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left(\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} \right)^2 \tag{3.22}$$

where m_t and v_t are the first moment (the mean) and the second moment (the variance) respectively of the gradients. We can see that m_t is a similar term to

momentum.
Since m_t and v_t are initialized to zeros, they are biased toward that value and it is needed to counteract this bias by computing the corrected moment estimates as:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1} \tag{3.23}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2} \tag{3.24}$$

and the update rule becomes:

$$\mathbf{W} \leftarrow \mathbf{W} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{3.25}$$

There are many others optimizers (Adagrad, Adadelta, RMSprop...) which work in similar ways, but currently Adam is one of the most used in practice given its very good performance across many different types of training tasks.

There are also various additional strategies that can be used to further improve performance of the training:

- *Batch normalization* [21]: during training, the parameters are updated all together in a single step, but the update of the parameters in a following layer is based on the parameters of the previous layer which are going to be updated too. This makes the training as forever chasing a moving target. Batch normalization was proposed to overcome this limitation and it does so by standardizing the activations of each layer, so that the following layers will receive roughly the same distribution of inputs. This technique improves the training by speeding it up and stabilizing it.
- *Early stopping*: it is a technique that monitors the training and validation set errors. If the validation error does not improve, or even get worse during training for some extended period of time, the training is stopped and the previous best possible model on the validation set is taken.
- *Gradient noise*: it consists in adding noise that follows a Gaussian distribution to every gradient update. The Gaussian distribution is centered in zero and the variance follows a specific schedule which helps the model to be more robust to bad parameter initialization. It could be due to the fact that it is easier to escape and find a new local minimum during training.

3.2 Recurrent neural networks

Recurrent Neural Networks (RNNs) are a type of neural networks introduced and developed in order to handle different-length sequential data. That is because

standard neural networks have difficulties in dealing with variable length data while RNNs overcome this issue. This is possible thanks to a recursive processing unit combined with a hidden state derived from the past. The use of an hidden state allows the information to flow through the sequence. In a *vanilla RNN*, the general formulas to compute the hidden state h_t and the output vector y_t are:

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h) \tag{3.26}$$

$$y_t = \sigma_y(W_y h_t + b_y) \tag{3.27}$$

Where:

- x_t is the input vector
- W_h, U_h are the weight matrices applied to the input and previous hidden state respectively
- b_h and b_y are the bias terms
- W_y is the weight matrix applied to the hidden state to compute the output
- σ_h and σ_y are the non linear activation functions

An illustration of a general RNN framework is displayed in Figure 3.5. From the recurrent pointing structure on the left of the figure we can observe that:

- It does not depend on the length of the sequence (i.e. we are not bounded to any specific length of the input/output).
- The hidden state represents the past sequence up to that point. To generate the future hidden state, the previous one is fed to the recurrent block together with the current input.
- All the parameters (weight matrices, biases and activation functions) are shared between the RNN cells.

Backpropagation Through Time

Backpropagation Through Time [23, 24] is the algorithm used to train RNNs. It is the same concept as backpropagation since we are still applying the chain rule to compute gradients, the only different is that at each layer the parameters to optimize are the same. The goal is to optimize the matrices W_h, U_h, W_y and the biases in order to minimize the objective function $L(y, \hat{y})$. The chain rule is applied starting from the last step to the first one, considering one step at a time. The

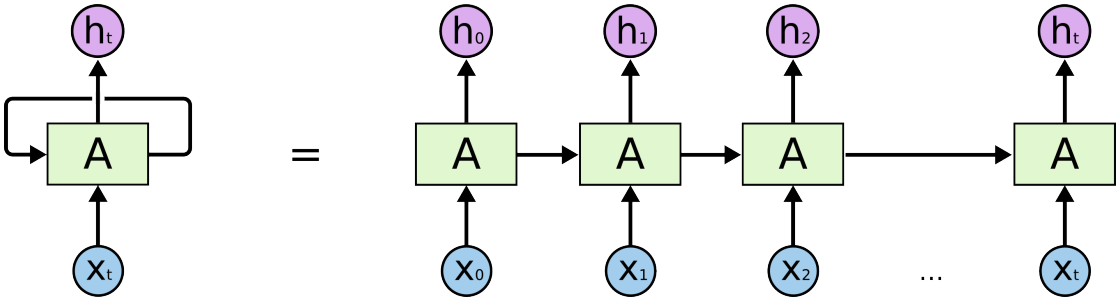


Figure 3.5: On the left a general RNN structure. On the right the equivalent unfolded version where every time step is represented with the same repeated structure. x_t and h_t are the input token and relative hidden state at time step t respectively. In the figure, the passage from h_t to y_t it's not represented (from [22]).

drawback using this kind of parameters update is that we cannot take advantage of any parallelization given the fact that we need to compute the gradients recursively, and the chain can get very long when t is large.

Advantages and problems of RNNs

Standard RNNs have some difficulties when dealing with long sequences, the main problem is that they cannot capture long term dependencies in the inputs when the sequence is long. This problem was investigated in [25, 26] that showed that this behaviour is caused by vanishing and exploding gradients. We have vanishing gradients in deep neural networks when the value of the gradients gets smaller and smaller as we propagate backward the gradients. In fact, if the gradients are very small, the parameters of the network do not change, making the training slow and inefficient.
On the other hand, RNNs can be considered very space efficient models since their dimension do not depend on the dimension of the input or output.

3.2.1 LSTM

Long Short Term Memory (LSTM) networks [27] are a kind of RNNs that overcome some limitations of the *vanilla RNN*. As the name suggests, they were designed specifically to tackle the long term dependency problem [25], making it possible to learn and remember information for long periods of time.
The key idea behind the LSTM is the cell state which is the top line in the right Figure 3.6. This line runs through all the network, easily carrying information from previous states. The LSTM has the abilty to add or remove information from the

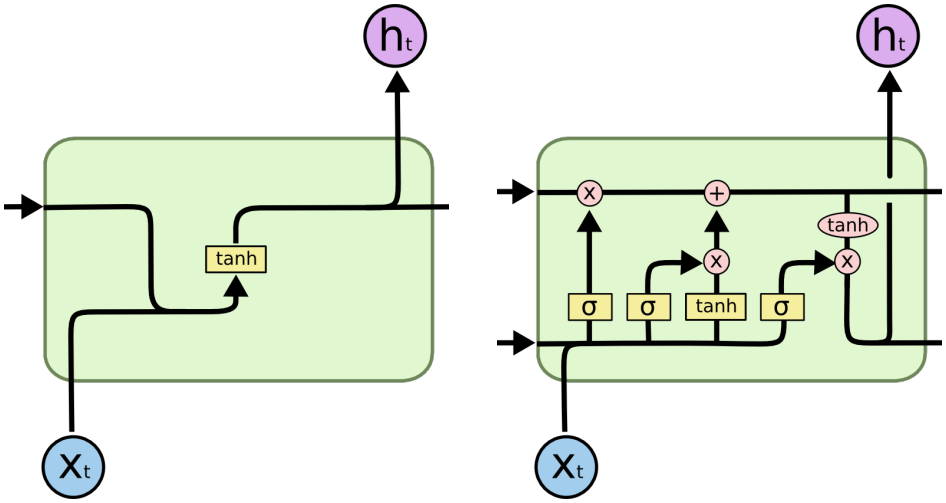


Figure 3.6: RNN cell architecture (left) vs LSTM cell architecture (right). From [22].

cell state through the gates. There are three gates in a LSTM cell:

- The forget gate G_f
- The update gate G_u
- The output gate G_o

The forget gate decides how much information retain from the previous cell state c_{t-1}. The decision is done based on the previous hidden state h_{t-1} and current input x_t passed through a linear transformation and a sigmoid function which outputs a number between 0 (forget) and 1 (remember). The operation of the forget gate G_f is:

$$G_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{3.28}$$

The update gate decides what new information we want to store in the cell. The first sigmoid layer decides which values of the cell state c_{t-1} we want to update, and then a tanh layer produce the values which we are going to add to the previous cell state. Thus, the overall update gate function is:

$$G_u = \sigma(W_i[h_{t-1}, x_t] + b_i) \odot \tanh(W_c[h_{t-1}, x_t] + b_c]) \tag{3.29}$$

Where $\odot$ represents the element-wise product. The next cell state vector c_t is then obtained with:

$$c_t = G_t \odot c_{t-1} + G_u \tag{3.30}$$

Finally, we have to derive the next hidden state h_t. The next hidden state is obtained through the current new cell state c_t and the concatenation of previous hidden state h_{t-1} and input x_t. We define the output gate G_o as:

$$G_o = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{3.31}$$

and consequently the new hidden state is:

$$h_t = \tanh c_t \odot G_o \tag{3.32}$$

The sigmoid function of the output gate decides which elements of the new cell state we are going to output.

3.3 Attention

The attention mechanism emerged first in the NLP and machine translation areas. The reason for it is that these are problems dealing with time-varying data and sequences (sequence to sequence models). Now, it has been extended to other domains too, such as computer vision, but the main approach is always to deal with sequences.
Traditionally, RNN based models that use an encoder-decoder structure are used when dealing with sequences. That is because RNN are sequential models and we want to transform an input (encoder) to an output (decoder). Both the encoder and decoder usually consist in stacked RNN layers, such as LSTM's. In basic machine translation, for example, each word of the input is fed to the encoder sequentially, together with the hidden vector from the previous iteration. This repeats until the last word is processed, and we obtain the encoded sequence representation c, the context vector. Usually, the encoding c has a fixed size and corresponds to the last hidden state of the RNN encoder. This context vector is then fed to the decoder which outputs autoregressively, one word at a time.
This approach however is not optimal for two reasons:

- *Bottleneck problem*: the context vector c has a fixed length so the information of the input that it can carry is limited.
- *Information loss*: RNN's have trouble dealing with long sequences. They have difficulties remembering information from far behind steps, so, in a long input sequence, the context vector c would carry much more information about the last steps than long past ones.

Attention was proposed to solve these issues. The main idea is that the decoder should have access to different parts of the input at different steps, not just to a context vector that represents all the input sequence. In other words, it should be

able to focus on different aspects at different steps, it should learn an *alignment* between target tokens and source tokens. The concept was introduced by Bahdanau et al. [28] in 2015. However, in the next subsections we describe the taxonomy of attention and the different types.

3.3.1 Global or local attention

The difference between global and local attention is how much of the input we want to concentrate on. In global attention, the attention is computed over the entire input sequence, the context vector is produced using all the hidden states of the encoder. Even though this approach is preferable, for long sequences in input it can become impractical and too expensive computationally speaking (for each output token, compute attention over all the input sequence). For this reason, local attention was proposed [29]. In local attention, the model learns an aligned position p_t for each target token at time t. The context vector c_t is then derived using a window of width $2D$ centered on p_t over the input tokens. p_t can be either simply put as t or computed dynamically. So, in local attention we only compute an alignment between the target token and a subset of the inputs, drastically reducing computational requirements needing only $2D + 1$ weights per token. The two types are represented in Figure 3.7.

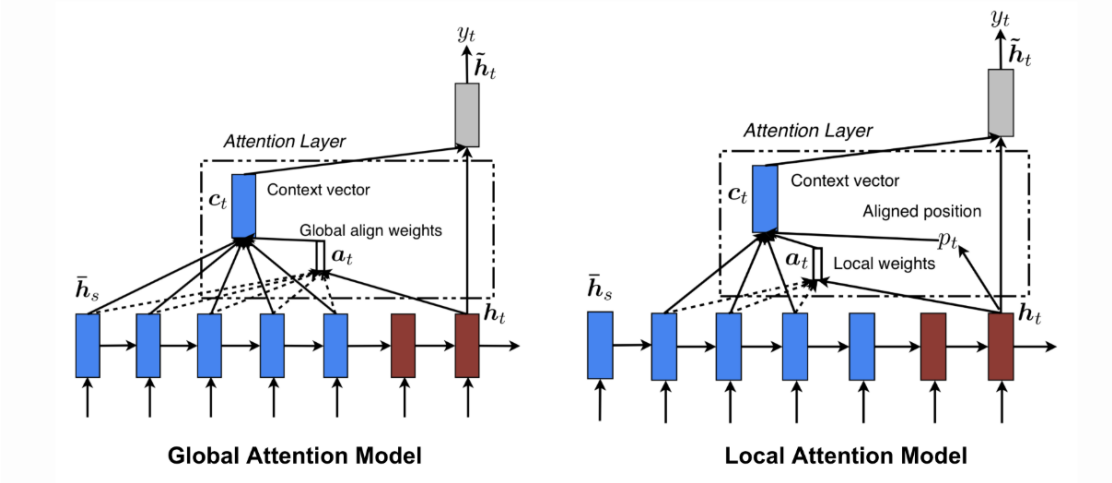


Figure 3.7: Global and local attentions from [29]. Each blue box is an input token hidden state, while the grey boxes are the target tokens states after transformation from attention. We can observe that the context vector c_t is computed with different source tokens between global and local attention (not the same connections to the context vector).

3.3.2 Hard attention

With hard attention we refer to a type of attention which is parametrized by non differentiable functions. It usually means that this type of attention uses a stochastic sample model and it is not trainable with standard gradient descent. At each time t, an hidden source state is sampled from a distribution (the sampled hidden state usually is a well aligned state). Using this sampling strategy, gradient descent is not usable and reinforcement learning techniques are used to train the models, making the training more difficult and not straightforward with respect to the soft attention counterpart.
Hard attention can be viewed as a switch mechanism to determine whether to attend a region or not, only focusing on a single token.

3.3.3 Soft attention

Soft attention is the one proposed first by [28] and differently from hard attention it is differentiable, thus standard backpropagation algorithms can be applied. It is a type of global attention and it consists in a weighted sum between all the encoder hidden states to represent the context vector at time i:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{3.33}$$

Where c_i is the context vector, T_x is the source sequence length, α_{ij} is the weight between target state at position i and source state at position j and h_j is the encoder state at position j. We can notice that the context vector is different for every target word.
The weights α are computed as:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \tag{3.34}$$

where

$$e_{ij} = \text{score}(s_{i-1}, h_j) \tag{3.35}$$

is called *alignment model* and measures how well the inputs at position j match the output at position i. s_i is the decoder hidden state at position i. It is interesting to notice that the weights are computed using a softmax function, which can give us a probabilistic interpretation. In fact, the probability α_{ij} reflects the importance of the hidden state h_j with respect to the previous decoder hidden state s_{i-1} in producing the next state s_i. By letting the decoder have the possibility to attend to all encoder states, we relieve the encoder from the problem of encoding all the

input sequence to a fixed-length vector.
There are various alignment models that can be exploited such as:

$$\text{score}(s_i, h_j) = \begin{cases} \text{cosine}[s_i, h_j] & \text{content-based} \\ v_a^T \tanh(W_a[s_i; h_j]) & \text{additive} \\ s_i^T W_a h_j & \text{general} \\ s_i^T h_j & \text{dot-product} \end{cases} \tag{3.36}$$

where v_a and W_a are weight matrices to be learned and [;] represents concatenation. The one proposed by [28] is the additive score, which consists in a small fully connected neural network (backpropagation available) and it is still used.
This kind of attention is the most used, but we should know that with the improvements it brings, it also has the downside that adds more computational complexity to the model. Examples of word alignments of soft attention can be viewed in Figure 3.8, while the practical difference between soft and hard attention in Figure 3.9.

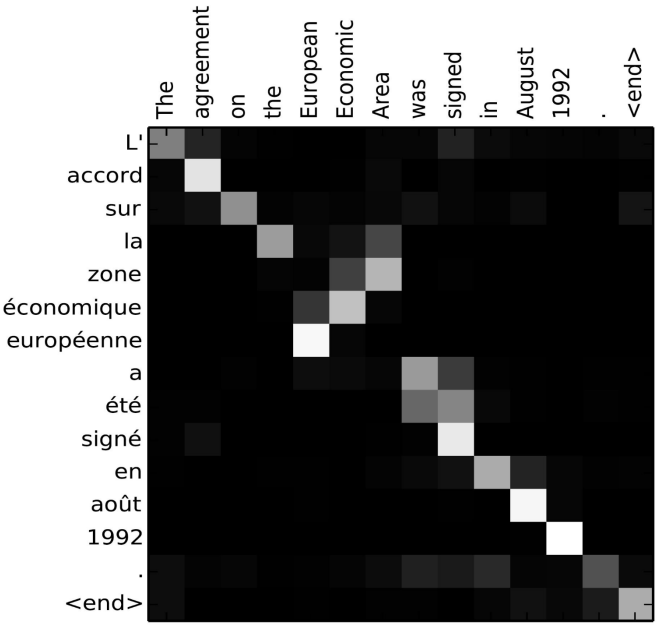


Figure 3.8: Example of soft attention on a English to French translation task. The brightness of the squares indicates the attention weight between the two words during the generation phase. It is interesting to notice that the model learned to align correctly the adjectives which are used in different orders from English to French (from [28]).

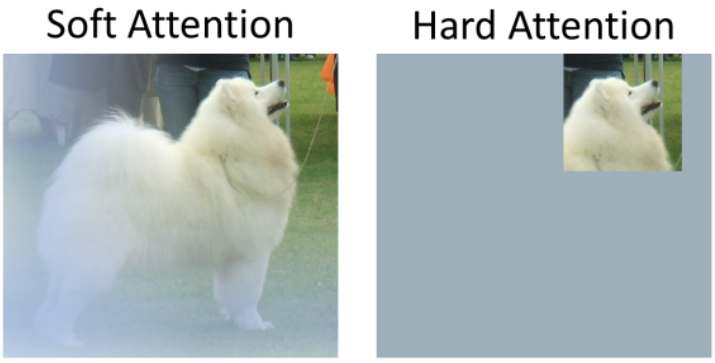


Figure 3.9: Soft vs hard attention. Soft attention is a global attention differentiable over the entire input domain, while hard attention focuses only on a state of the input sequence and it is a local attention to only one state, it typically requires reinforcement learning techniques to be trained.

3.3.4 Self attention

Self attention, also referred to in literature as intra-attention, is an attention mechanism relating different positions of a single sequence. It has been used in machine reading, abstractive summarization, image description generation and it is a key component of the Transformer architecture as we are going to see later in this work. In machine reading for example, self attention let us estimate a correlation between words in a text, scoring the alignment similar to as previously mentioned. The end goal of this type of attention is to create a meaningful representation of the sequence before transforming it to another one.

3.4 Transformers

The Transformer is a novel architecture presented in 2017 by Vaswani et al. [6] that deals with sequences. It was firstly applied to the machine translation task (in the original paper English-to-German and English-to-French) and then it was generalized to many other tasks, including computer vision. The novelty in the architecture stands in the fact that it totally replaces recurrences and convolutions, relying solely on the attention mechanism we have seen above to draw global dependencies between input and output. The name is *Transformer* in fact, because it *transforms* the inputs gradually using the attention mechanism, to obtain the desired outputs. The total replacement of recurrences gives a significant speed-up in training, since now the model can benefit from parallel training, achieving

state-of-the-art results in many tasks. Another benefit with respect to RNNs is that it is able to model long-term dependencies between the inputs, replacing the backpropagation-through-time with standard backpropagation as well as attention mechanism, making it possible to overcome some issues linked to the vanishing gradient problem.

3.4.1 Transformer architecture

As most of the sequence-to-sequence models, the Transformer model use an encoder-decoder architecture. In these sections we are going to refer to the standard Transformer model, the one of [6].
The Transformer architecture can be seen in Figure 3.10. First, the encoder maps an input sequence of symbol representations $(x_1, ..., x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, ..., z_n)$ called encoded representation. From this encoded representation, the decoder generates the output sequence $(y_1, ..., y_m)$ one element at a time, in an autoregressive fashion. This means that at each step, the decoder consumes the previously generated symbol as additional input for the next generation.

Interestingly, as we can see from the figure, the generation of the words in the decoder comes from different self-attention layers, that combine both the previously generated output and the encoded representation of the input. That is because, when decoding a sequence, we want to pay attention both to the source and to the partial current state of the output.
To understand the Transformer, we need to present the different parts which is composed of, that are:

- The attention mechanism
- The encoder structure
- The decoder structure
- The input embeddings
- The positional encoding

3.4.2 Attention mechanism

The attention mechanism is one of the most important components in the Transformer. As already described in the previous section, attention can be seen as a way to align two sequences, making possible to focus more on certain aspects of the input sequence instead of the whole. In fact, an attention function can be described as mapping a query and a set of key-value pairs to an output, where the

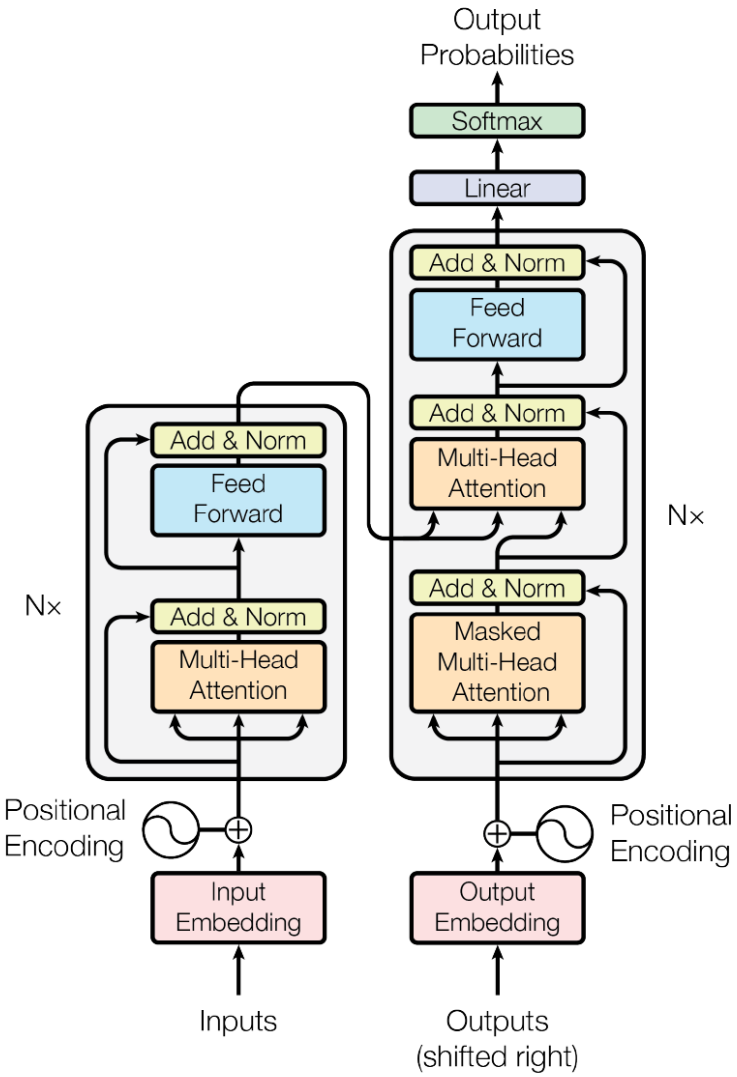


Figure 3.10: Transformer architecture as proposed in [6]. The grey block on the left is the encoder block, while on the right is the decoder block.

query, keys, values, and output are all vectors [6]. The output is a weighted sum of the values, where the weights are computed using a compatibility function of the query we are searching with the corresponding key.
The type of attention used in the Transformer is the *scaled dot-product attention* (Figure 3.11).

The inputs of the attention are vectors of queries and keys of dimension d_k, and values of dimension d_v. For conciseness and simpler notation, set of queries,

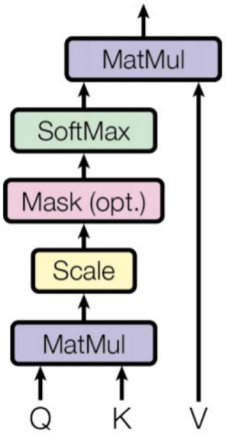


Figure 3.11: Scaled dot-product attention operations (from [6])

keys and values are stacked together and represented by matrices Q, K and V respectively. The dot product of the queries with all the keys is computed, then divided by $\sqrt{d_k}$ and then passed through a softmax function to obtain the weights of the values. Using the matrix notation we can write:

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{3.37}$$

The term $\sqrt{d_k}$ is used for numerical stability reasons, the authors suggested that if the input sequence is large, the dot product could grow large in magnitude, pushing the softmax function to values that have extremely small gradients. The square root term was added to counteract this problem. The softmax function instead is used to have a sum to one for each query.
In practice, we score the queries with the keys to get a compatibility value (weight) which is used to weight the corresponding value vector.

This is the concept used in all the attention blocks of the Transformer, but what are queries, keys and values? How do we compute them?
Queries, keys and values are projections of the previous layer embeddings. These projections are just the multiplication of the inputs with matrices learned during the training phase. The main difference between the various attention blocks is in what is the input projected as keys, queries and values. For this reason there are three types of attention blocks.

Encoder self-attention block

In the encoder self-attention block, all queries, keys and values come from the same input, the source sequence. The goal of this type of attention is to transform the input embeddings by allowing the model to look at other positions in the input sequence for clues that can help lead to a better encoding for the token [30]. In this case, the attentional weights are computed over the whole input sequence, since each position can attend to all positions of the previous layer. For example, if we are in an NLP task and we are encoding the sentence

The animal didn't cross the street because it was too tired.

How does the machine know that the word *it* refers to the animal? It does so thanks to the encoder self-attention. The encoding of the world *it* is transformed in the various layers to encode also the information of what *it* refers to.
It creates a meaningful representation of the input by learning the relations and syntax of the source sequence.

Decoder self-attention block

The decoder self-attention block is very similar to the encoder one. The only differences are that the queries, keys and values are all obtained from the decoder input sequence, instead of from the source sequence, and that each positions cannot look to all the other positions in the attention layer. Each position in the decoder is allowed to attend only up to and including the considered position. It is important to preserve the autoregressive property and this is done by masking subsequent ground truth words during training, forcing to zero their weights in the attention pass.

Encoder-decoder attention block

In the encoder-decoder attention block, the second attention module in the decoder, we have the queries which are computed from the previous decoder layer, while the keys and values come from the output of the encoder. In this way, we can score each query, so each position in the decoder, over every position of the input sequence. This type of attention is the most similar to the traditionally used one previous to Transformers in sequence-to-sequence models, the one proposed by Bahdanau and explained in Section 3.3.

Multi-head attention

Multi-head attention was introduced to expand the model's ability to focus on different positions while keeping a very similar computational cost to the single head

counterpart. It consists in projecting h times the queries, keys and values, reducing their dimension from d_{model} to d_k, d_k and d_v respectively, and then performing the scaled dot-product attention pass h times with the h different projections. We then concatenate the h d_v dimensional output values and finally project the result again to obtain the final attentional output. The process is represented in Figure 3.12. Formally we can write:

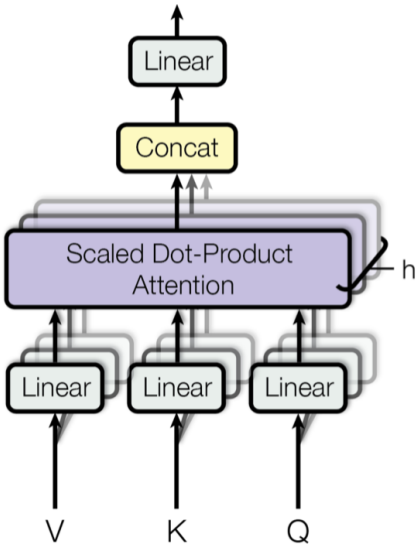


Figure 3.12: Multi-head attention as in [6]

$$\begin{aligned} \text{MultiHead}(Q,K,V) &= \text{Concat}(\text{head}_1,...,\text{head}_h)W^O \\ \text{head}_i &= \text{Attention}(QW_i^Q,\, KW_i^K,\, VW_i^V) \end{aligned} \tag{3.38}$$

Where $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$. In the original paper by Vaswani, h is set to 8, $d_{model} = 512$ so $d_v = d_k = 64$. Using this approach, it gives the attention layer multiple *representation subspaces*, meaning that by segmenting the input embedding across multiple heads, different sections of the embedding can attend different per-head subspaces. It can almost be seen as a form of ensembling.

3.4.3 Encoder

The encoder in the original Transformer is composed by a stack of $N = 6$ identical layers, where each layer of the stack is composed by two sub-layers. To each one of the two sub-layers, a residual connection is employed, followed by a layer

normalization. That is, the output of each sub-layer is:

$$\text{LayerNorm}(x + \text{Dropout}(\text{Sublayer}(x))) \tag{3.39}$$

where $\text{Sublayer}(x)$ is the function of the sub-layer itself.
There are two kinds of sub-layers: a multi-head self-attention mechanism and a simple fully connected feed forward network (FFN). The self-attention sub-layer is explained in the above section, while the fully connected one consists only in two linear transformations with a ReLU activation in between. Formally:

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{3.40}$$

The N layers are identical in structure, but do not share weights.

3.4.4 Decoder

In the originally proposed Transformer, also the decoder is composed by a stack of $N = 6$ identical layers. But, differently from the encoder, each one of the decoder layers has three sub-layers. The inputs of the decoder, which are the decoded tokens after i steps (recalling the auto-regressive property of the Transformer), pass at first through a masked multi-head self-attention sub-layer, then through a multi-head cross-attention sub-layer and finally through a fully connected feed-forward sub-layer. Similarly to the encoder, for each sub-layer a residual connection is employed followed by a layer normalization.
There are two attentional sub-layers because the first one is used to generate a meaningful representation and to look for relations in the decoder inputs, while the second, the cross attention sub-layer, is used to relate the encoder and the decoder, allowing the decoder to focus on the parts of the input on which it wants to focus more, thus conditioning the generation process.
The masking, together with the fact that the output tokens are offset by one position, ensures that the generation for position i can depend only on the known outputs at positions less than i.

3.4.5 Input embedding

In order for the model to process the input information, it must be in a numerical form. In the case of the original Transformer, since we are dealing with words, we need to transform words into numbers prior to inputting them to the model, but the practice of embedding the input is present in other situations too. For example, even if we have (x, y) coordinates or velocities in the case of trajectory forecasting, and they are in numbers, we need to embed this information in a standard and understandable way for the model.

So, if we are dealing with text, a technique called *word embedding* is used to map words or phrases from a vocabulary to a corresponding vector of real numbers. This representation is both more efficient and more expressive. It is different from a *bag of words* approach, where every word is represented with a one-hot encoded vector of the vocabulary dimension. This representation is very sparse and has a very high dimensional space. Furthermore, a bag of word approach does not take into account any information about the word, the semantic information, while instead using a word embedding preserve these similarities, and semantically similar words are close in the vector space.
There are different algorithms which are used to find word embeddings where the most popular are word2vec [31] and GloVe [32]. In the case of the original Transformer, the embedding vectors have dimension $d_{model} = 512$.

3.4.6 Positional encoding

One last key component of the Transformer is the positional encoding. It is essential for a model that deals with sequences to know the order of the inputs, this is important for a language task (words have an order in the sentence), but also in general to any kind of sequence task. RNNs by design take into account the order of the sequence, processing it in a sequential manner, but Transformers do not have this characteristic using only attention without any recurrence. Since Transformers can benefit from parallel training and every input token flows simultaneously through the model, they need to know the position of each one of them. In order to know each token position, Vaswani et al. proposed the positional encoding, adding this information to every input embedding.
We could think that assigning a number between [0,1], where 0 is the first token and 1 the last one, could be a valid idea, but in this way the model would not be able to figure out how many words are present in a specific range. Different length sequences would have different deltas between tokens.
Another idea could be to assign increasing numbers from 1 onward to the tokens. The problem here is that numbers could grow large and the model would not generalize well without seeing samples of all lengths [33]. We need an encoding such that:

- It should output a unique encoding for each time-step.
- Distance between any two time-steps should be consistent across sequences with different lengths.
- It should generalize to longer sentences without any efforts
- It must be deterministic.

The authors proposed a solution which is not a single number but a d_{model} dimensional vector, where d_{model} is the dimension of the input embedding. Formally:

$$\begin{aligned} PE_{(pos,2i)} &= \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \end{aligned} \tag{3.41}$$

where pos is the position and i is the dimension. Each dimension of the positional encoding corresponds to a sinusoid. We can visualize the positional encoding in Figure 3.13 and we can observe that the frequencies are decreasing along the vector dimension.

An important property of the proposed positional encoding is that it allows the

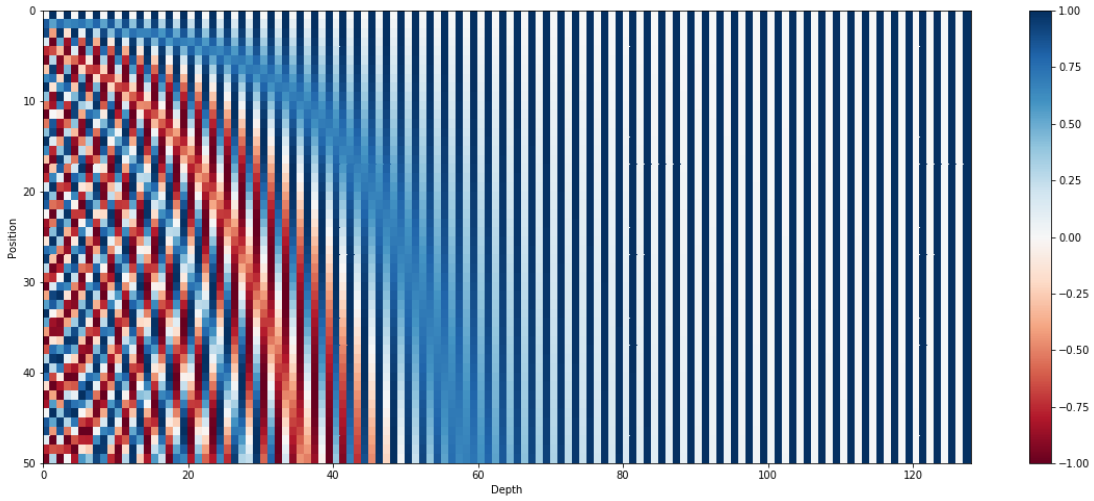


Figure 3.13: First 50 positional encodings using a d_{model}-dimensional vector of 128. Each row represents a positional encoding vector. From [33].

model to attend to relative positions. The authors hypothesized this, because for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.
The positional encoding is then summed with the input embeddings (this is done only once before the first encoder or decoder layer) to inject the positional information.

3.5 Generative models

3.5.1 GANs

Generative Adversarial Networks [34], abbreviated as GANs, are a class of *generative models*. Generative models are a type of models which try to replicate and estimate a probability distribution p_{model}, from samples drawn from a distribution p_{data} of the training set. In order to do so, GANs use a game between two players. The first player is called the *generator* G and the second the *discriminator* D. The goal of the generator is to generate samples that resemble the samples of the training set and that are indistinguishable from it, while the goal of the discriminator is to tell whether the samples which are fed to it are real or fake, fake meaning that they were created by the generator. To succeed in the game, the generator must fool the discriminator. Both the generator and the discriminator are differentiable functions which usually are neural networks.
So, the generator tries to capture the data distribution and a discriminator estimates the probability of that sample. To learn the generator distribution, the generator builds a mapping function from a prior noise distribution $p_z(z)$ to $G(z;\theta_g)$, while the discriminator only outputs a scalar $D(x;\theta_d)$ and we simultaneously train both. The parameters of the models are updated by minimizing $\log(1 - D(G(z)))$ for G, and maximizing the probability of assigning the correct label for D. D and G play a minimax game with value function $V(G, D)$:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \quad (3.42)$$

An important thing to notice is that generative models are stochastic by nature, meaning that they could learn to represent multi-modal distributions. This is very useful in various tasks, such as image generation, next frame prediction, etc., but in trajectory forecasting too. In fact, even if we have a single ground truth trajectory, there are multiple possible ways to overcome an obstacle for example, or to avoid collision with other pedestrians. Since it is intrinsically stochastic, the pedestrian trajectory prediction task can be effectively addressed with generative models.
However, there is a problem when using vanilla GANs for trajectory forecasting, the fact that in simple GANs the input to the generator is random noise, and that there is no control in what is going to be the output. How can we decide what should be the output? Conditional GANs come into play.

3.5.2 CGANs

Conditional GANs (CGANs) [35] were proposed in order to have more control on the sample generation process. In this case, both the discriminator and the generator are conditioned with some extra information y. y can take any form

and it is not bounded by being only the class label for example. The conditioning is done by composing the prior noise $p_z(z)$ with y, and it can be done in several different ways. An overview of the CGAN architecture is illustrated in Figure 3.14.

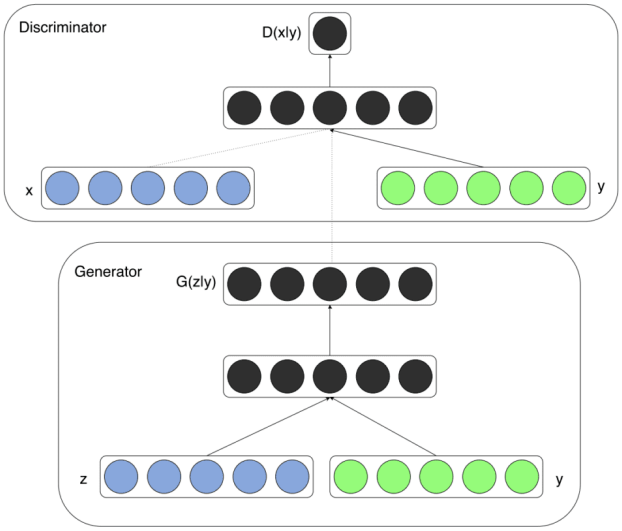


Figure 3.14: Conditional GAN architecture as proposed in the original paper [35]. Both the discriminator and generator are additionally fed the y conditioning (in green). During training, alongside the y conditioning, the input to the discriminator can be either the generator output or the sample from the training set (light grey lines).

The objective function of the minimax game conditioned on y can be expressed as:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x|y)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z|y)))] \quad (3.43)$$

3.5.3 Variational Autoencoders

Variational Autoencoders (VAEs) [36] are a class of generative models which derive from autoencoders. Autoencoders are networks that try to compress the input to a lower dimensional space (called *latent space*), which is then reconstructed to the original input dimension. The lower dimensional vector is also called the encoding vector or the latent vector and each dimension of it represents a feature of the data. Differently from a simple autoencoder, a variational autoencoder provides a probabilistic formulation of the latent space, where each attribute of the latent vector is no longer described by a single value but by a probability distribution.

For example, if an autoencoder is trained on a large dataset of images of faces with a small encoding dimension, each dimension of the encoding would learn descriptive attributes of a face, such as skin color or smile, trying to compress the initial image into a low dimensional representation. Each attribute of the encoding vector would be a single value, but we would like to have a probabilistic representation of the input, where each of the latent attributes follows a probabilistic distribution and can have a value in a range of values. Once we have the encoded representation where each of the attributes is described by a probability distribution, we sample from them and use the sampled values as inputs to the decoder. By not having just a single-valued latent vector, but a more general statistical distribution, we are enforcing the model to have a continuous and smooth latent space representation, where nearby samples should have almost identical results [37].

Mathematically, we define z as our hidden variable and x as our observation. We would like to compute the posterior $p(z|x)$ because the goal is to infer good values of latent variables given observed data. The posterior can be rewritten as:

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)} = \frac{p(x,z)}{p(x)} \tag{3.44}$$

but computing $p(x)$ (called evidence) is intractable since

$$p(x) = \int p(x|z)p(z)dz \tag{3.45}$$

and would require exponential time to evaluate all possible configurations of the latent variables. So we need to approximate the posterior distribution $p(x|z)$. We can do it by introducing a new distribution $q(z|x)$, where we can choose a tractable distribution (e.g. Gaussian) and play with its parameters in order to approximate the true posterior $p(z|x)$. To approximate the posterior we need a measure of dissimilarity between the two distributions $q(z|x)$ and $p(z|x)$, and this measure is the Kullback-Leibler divergence defined as:

$$\mathbb{KL}(q||p) = -\sum q(x) \log \frac{p(x)}{q(x)} \tag{3.46}$$

We need to minimize this divergence between $p(z|x)$ and $q(z|x)$, so, in our case, the KL divergence is rewritten as:

$$\mathbb{KL}(q(z|x)||p(z|x)) = -\sum_z q(z|x) \log \frac{p(z|x)}{q(z|x)} \tag{3.47}$$

and, by substituting 3.44, we obtain:

$$\begin{aligned}
\mathbb{KL}(q(z|x)||p(z|x)) &= -\sum_z q(z|x) \log \left(\frac{\frac{p(x,z)}{p(x)}}{q(z|x)} \right) \\
&= -\sum_z q(z|x) \log \left(\frac{p(x,z)}{q(z|x)} \frac{1}{p(x)} \right) \\
&= -\sum_z q(z|x) \left[\log \frac{p(x,z)}{q(z|x)} - \log p(x) \right] \\
&= -\sum_z q(z|x) \log \frac{p(x,z)}{q(z|x)} + \sum_z q(z|x) \log p(x) \\
&= -\sum_z q(z|x) \log \frac{p(x,z)}{q(z|x)} + \log p(x) \sum_z q(z|x) \\
&= -\sum_z q(z|x) \log \frac{p(x,z)}{q(z|x)} + \log p(x)
\end{aligned} \tag{3.48}$$

where, in the last passage, we brought out of the summation $\log p(x)$ since it does not depend on z, and $\sum_z q(z|x) = 1$. Finally, based on the previous derivations, we can write:

$$\begin{aligned}
\log p(x) &= \mathbb{KL}(q(z|x)||p(z|x)) + \sum_z q(z|x) \log \frac{p(x,z)}{q(z|x)} \\
&= \mathbb{KL}(q(z|x)||p(z|x)) + ELBO(\lambda)
\end{aligned} \tag{3.49}$$

where λ are the parameters of the chosen posterior distribution. From the above equation we can observe that $p(x)$ is constant, independently from q and how we choose to parametrize the posterior distribution. From this observation we can change the objective function, that instead of minimizing KL (always greater than zero and intractable since it requires the computation of $p(x)$) becomes maximizing the ELBO (Evidence Lower BOund) which is computationally feasible. This is possible since $p(x)$ is constant and minimizing KL is equivalent to maximize the ELBO. It is called this way because it is a lower bound to the actual value of $\log p(x)$ which is always going to be greater or equal. We can then rewrite ELBO as:

$$\begin{aligned}
ELBO &= \sum_z q(z|x) \log \frac{p(x|z)p(z)}{q(z|x)} \\
&= \sum_z q(z|x) \left[\log p(x|z) + \log \frac{p(z)}{q(z|x)} \right] \\
&= \mathbb{E}_q[\log p(x|z)] - \mathbb{KL}(q(z|x)||p(z))
\end{aligned} \tag{3.50}$$

To recap, in order to make q similar to p we need to minimize the KL divergence, which is equivalent to maximize the ELBO which is in the form of Eq. 3.50. That

means encouraging $q(z|x)$ to be similar to $p(z)$ (the prior) while at the same time maximizing the likelihood of the observation given z. So, the first term of 3.50 is the reconstruction error, while the second, differently from the simple autoencoder, forces the encoder output distribution to be similar to the prior $p(z)$ which is chosen (Gaussian).
In neural network terms, $q(z|x)$ is the encoder network, while $p(x|z)$ is the decoder network as represented in Figure 3.15. The concept is as adding a regularizing term to the autoencoder cost function so that the latent space has a certain chosen distribution. The encoder outputs the parameters of the chosen distribution (μ and Σ in case of a Gaussian chosen prior) and the latent vector z is sampled from the distribution which has the outputted parameters. To finally use the model as a generative model, we can drop the encoder part, sample directly from the prior distribution and feed it to the decoder to generate an observation.

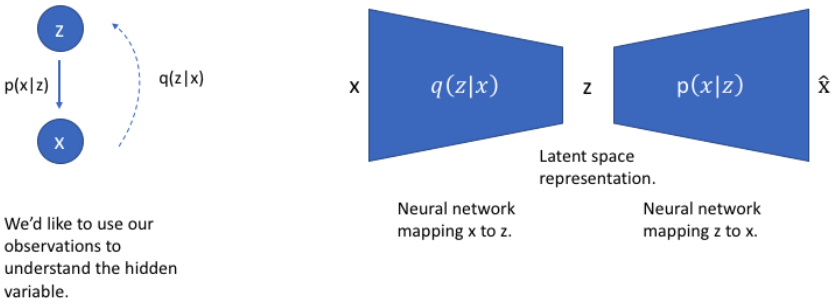


Figure 3.15: VAE architecture from [37]. Encoder and decoder are trained to maximize the ELBO. The encoder learns the mapping from x to z, thus $q(z|x)$, while the decoder learns the mapping from z to x, $p(x|z)$.

3.5.4 Conditional Variational Autoencoders

The problem with Variational Autoencoders is that we do not control the data generation process, we cannot decide to generate some specific data. For this reason Conditional Variational Autoencoders (CVAEs) were introduced [38]. With respect to VAEs, which model latent variables and data directly, CVAEs model latent variables and data conditioned to some random variables.
If we look at VAE objective function 3.50, we notice how the encoder models the latent variable z directly from the data x, not discriminating between the type of x. For example, if we are training the model on a labelled dataset, the model does not take into consideration the label of the data. The same can be said for the decoder since it outputs data only conditioned on latent variable z. To improve

the model and to be able to output specific types of data, we condition the encoder and decoder to another variable c and modify the objective function 3.50 to:

$$ELBO = \mathbb{E}_q[\log p(x|z,c)] - \mathbb{KL}(q(z|x,c)||p(z|c)) \tag{3.51}$$

In practice this modified version consists in feeding both the encoder and decoder with another variable c which could be anything, such as the label of the data. Now the prior $p(z)$ is conditioned to another variable c, so that for each possible value of c we would have a prior $p(z)$. The comparison between the VAE latent space and CVAE latent space can be observed in Figure 3.16.

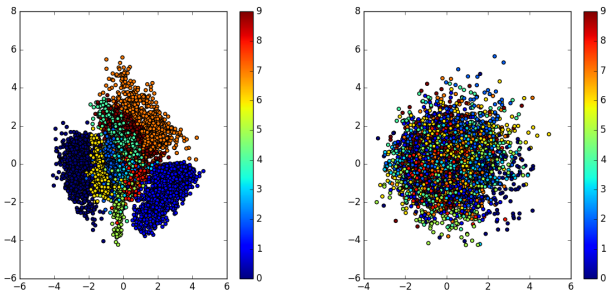


Figure 3.16: Latent variable z distribution in the case of a VAE (left) or CVAE (right). This example is from a model trained on MNIST dataset with a 2D latent vector dimension. Each color corresponds to a digit between 0 and 9. We can observe how in VAE latent space the different digits occupy different portions of the space and similar digits are close to each other. In the case of CVAE instead, we do not have any distinction between the digits in the overall latent space, but if we consider each digit distinctly, every distribution $q(z|x,c=c)$ follows a Normal distribution $\mathcal{N}(0,1)$.

Chapter 4

Deep learning models for trajectory forecasting

Prior to the advent of deep learning, the physics-based models illustrated in Section 2.4 were the most common approaches to tackle the pedestrian trajectory forecasting task. In recent years however, data-driven approaches have become widely used in this domain too. The data-driven approach consists in learning pedestrian trajectories directly from data, and it is useful in tasks where formulating the patterns in mathematical terms is very difficult. In our case, human motion is not easily predictable and easily modelable, with all the complex and subtle details of pedestrian motion. In fact, these new data-driven models outperformed the physics-based counterparts, proving that extrapolating rules of pedestrian behaviour directly from data can be a good approach.
In general, a global data-driven pipeline for trajectory forecasting is in the form of Figure 4.1. It usually consists of a motion encoding module and an interaction encoding module which output a social representation of the scene. The motion module encodes the past motion of pedestrians, while the interaction module captures the social interactions. These two modules are usually not mutually exclusive, meaning that some models take into account both aspects, while others just one of them. Then, once the social representation is available, it is fed to the decoder to predict a single trajectory (*deterministic*) or more trajectories/a trajectory distribution (*stochastic*) depending on the decoder architecture.
Various models have been proposed for the task, but some more than others have had more influence in the field. In the next sections some of the most important ones are presented, first by looking at RNNs based models, prior to Transformers, and then some more recent models which use this new architecture.

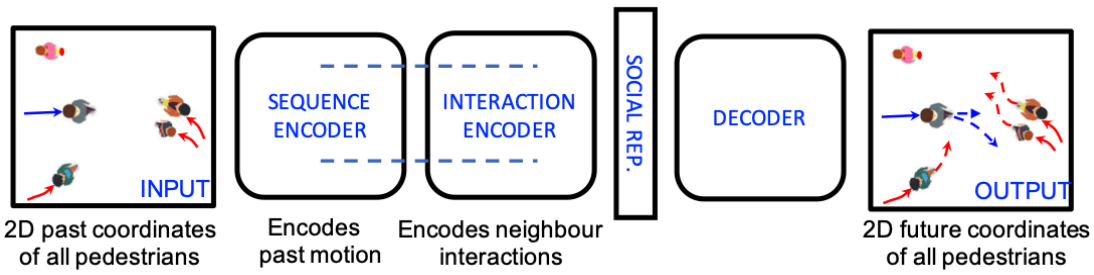


Figure 4.1: Standard data-driven pipeline for pedestrian trajectory forecasting. From [9].

4.1 RNN-based models

4.1.1 Social LSTM

The Social LSTM model [3], by Alahi et al., was one of the first deep learning models proposed for pedestrian trajectory forecasting. It was a pioneering work in the field, inspired by the success of RNNs in that years in sequence prediction tasks. It is a model which takes into account interaction between pedestrian in a novel way. The authors motivated the work by two reasons:

- Previous work used hand-crafted functions to model interactions.
- Previous work focused on avoiding immediate collisions without anticipating interactions that could occur in a more distant future.

In the model, each agent is modelled using a separate LSTM which is used to encode past history information, but how is it possible for the model to be interaction-aware? To be aware of the surroundings of each agent? In fact, LSTMs have the ability to learn sequences, but they do not capture dependencies between multiple correlated sequences, as trajectories are. This problem is addressed by adding a *social pooling layer* that allows agents close to each others to share their hidden states. This mechanism makes it possible to automatically learn interactions from the data itself. The overall model architecture is illustrated in Figure 4.2. The social pooling layer also needs to preserve the spatial information and to be able to cope with a variable number of agents, since the number of neighbors is not fixed. In order to deal with these challenges, a fixed dimensional grid is constructed around each pedestrian, and each cell of the grid is filled with the hidden states at that time of neighbour pedestrians. If two or more pedestrians are in the same grid cell, their hidden states are summed together. Having this compact and fixed representation for each pedestrian in the scene, the social tensor

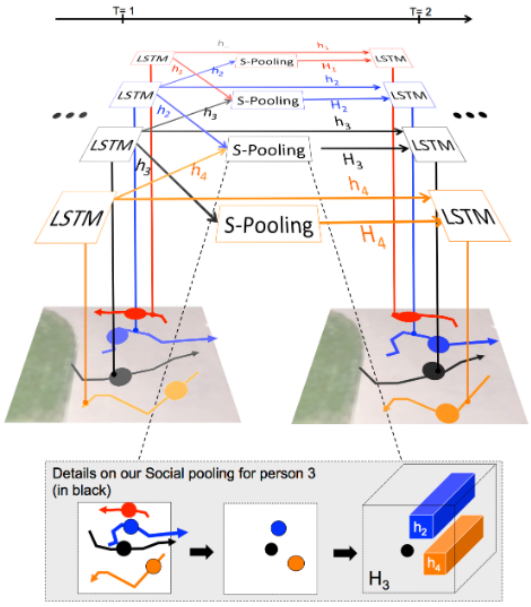


Figure 4.2: Social LSTM model from [3]. Separate LSTMs model each one of the agents and are interconnected by the social pooling layer which allows neighbor information to be encoded. In the bottom a detail of the social pooling layer for one person.

is embedded through a MLP, concatenated with the hidden state of the pedestrian considered and passed through the LSTM to generate the next prediction.
The next coordinates prediction is done by passing the hidden state through a linear layer, which outputs the parameters of a bivariate Gaussian distribution from which the actual coordinates can be sampled. In this sense, the Social LSTM model is not deterministic.
Lastly, the authors experimented also with a version that only pools the coordinates of neighbour pedestrians, not the hidden states, which in overall performs worse than the hidden states counterpart, due to the fact that hidden states encode more information about the history of the trajectory which can be used not only to avoid immediate collisions but also more distant ones.
From this work, the interest grew in NN-based solutions for trajectory forecasting, and the work has been extended to use also physical space context as in [39] or [40].

4.1.2 Social GAN

Social GAN [4] followed the work of the Social LSTM model and extended it by using a generative approach with GANs. The work was motivated by the fact that existing methods at the time tended to learn an average behaviour and did not make any effort in modelling multiple socially accepted trajectories. The main contributions of the work are:

- The use of a Conditional GAN for trajectory forecasting.
- A new *variety loss* which encourages the model to learn multi-modal distribution.
- A new pooling mechanism based on Social LSTM [3].

The model architecture is illustrated in Figure 4.3. It consists in 3 principal

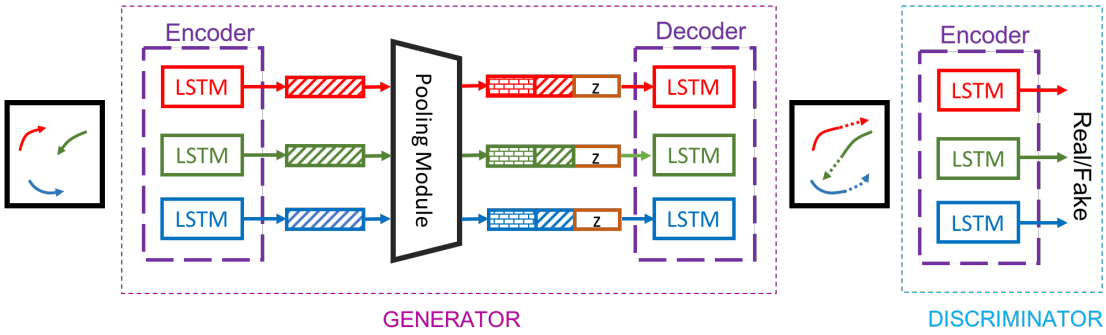


Figure 4.3: Social GAN architecture from [4].

components: the generator, the pooling module and the discriminator.
The generator is an encoder-decoder framework similar to [3] which, in order to condition the generation of trajectories to previous steps, concatenates the noise vector z to the output of the encoder layer and of the pooling module. Since the stochasticity of the model is introduced by the generative approach, the output of the decoder are simply the coordinates and not a bivariate Gaussian distribution. The discriminator instead is just an LSTM encoder that classifies trajectories as real or fake.
The pooling module deviates from prior work by discarding a grid-based approach and using instead the relative positions of neighbours. This approach can model all interactions between people and, instead of being used at each step, it is only used after the encoding, as an input to the decoder.
The last novelty introduced by [4] is the so called *variety loss*. The need of this new type of loss is because by using the standard L2 loss, the model tries to produce an average of the predictions where there can be more socially acceptable predictions.

It is a loss which can be useful when dealing with multi-modal outputs but with just one correct target. In this case, multiple predictions for the same past are produced by sampling different z and only the best prediction is used to update the network parameters:

$$\mathcal{L}_{variety} = \min_k \left\| Y_i - \hat{Y}_i^k \right\|_2 \tag{4.1}$$

where k is the number of predictions and it is arbitrary. The authors argue that using this loss, the model is encouradged to try to cover the space of possible and socially acceptable outputs that conform to the past trajectory used as conditioning.

4.1.3 Trajectron++

Trajectron++ [5] is a trajectory forecasting model that uses the CVAE framework. Its trajectory forecasting module consists in a complex architecture which does not only forecast human motion, but it can also predict the different motions of several types of agents, such as vehicles, bicycles, etc. The scene is represented as a spatio-temporal graph where each node represents an agent and each edge a connection between agents, meaning influence between the two. With respect to the above mentioned models, it can encode also the map information, allowing the model to make predictions based also on the street configuration. The model is illustrated in Figure 4.4. The agents and their history are modelled through

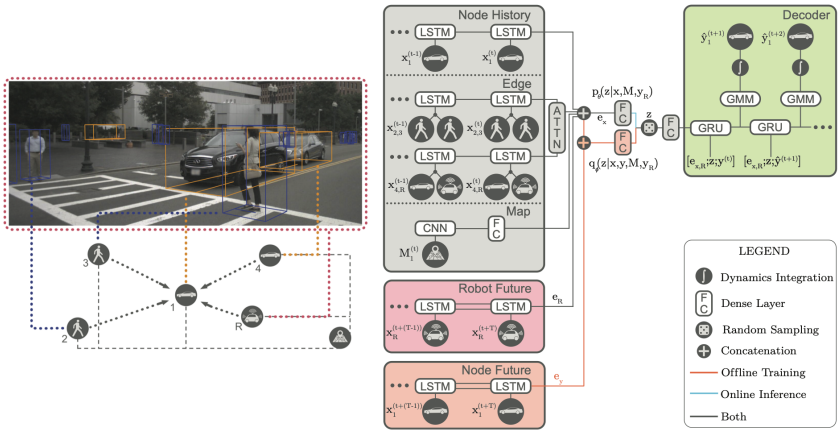


Figure 4.4: Trajectron++ architecture from [5]. In the gray box, the modelling of agent history at the top, the modelling of interactions in the middle and the inclusion of map information at the bottom. In orange the encoding of future steps used in training for the CVAE.

LSTMs and their interactions too. The semantic type of the agents is also encoded

in the LSTMs. The novelty is in the incorporation of new heterogeneous data which is done using a CNN.
In this case, the multimodality of the task is modelled through the CVAE latent variable framework. Since it uses a generative approach it is stochastic by nature, and this can be used to allow for different output modes. Multimodality is addressed by introducing a discrete categorical latent variable z that is used to model $p(y|x, z)$. From the figure we can observe how the CVAE framework is used for the task. The model during training is conditioned on the encoded future representation (which is not available at test time), and the latent variable z is obtained from both past and future encodings to learn the underlying conditioned distribution. At test time, z is sampled from the chosen prior instead, and fed, together with the encoded past history, to the decoder.

4.2 Transformer based models

4.2.1 Transformer TF

The Transformer TF (Trajectory Forecasting) [7] is one of the first proposed models in literature that make use of the Transformer architecture to forecast pedestrian trajectories. It is a fairly simple model since it does not require any particular modifications from the standard Transformer proposed by [6]. The overall architecture is in Figure 4.5. The main novelty introduced is the switch from LSTMs to the Transformers.
In contrast with previous models, Transformer TF does not include neither social information or map information, every pedestrian is modelled singularly using only its trajectory as the source for forecasting. Despite its simple approach, the model outperformed many of the other models which included also other inputs in addition to trajectory history. This shows the capability of Transformers to model sequential data versus the LSTM counterpart. A reason for this observed improvement in the trajectory forecasting task, as the authors suggest, could be that LSTMs accumulate in their hidden state both the encoded and decoded part, having to decide what to keep or forget, while Transformers keep the two parts separated choosing at each time on what to focus with the possibility of retrieving it. As stated before, the architecture does not have any major changes with respect to a standard Transformer, the input coordinates are fed to an embedding layer, then to a positional encoding layer. The positional encoding layer is important because, similarly to what it does in NLP models indicating the position of a token in a sentence, it encodes the timestep information of the coordinates.
Other than Transformer TF, the authors introduced two more models, BERT TF and Transformer TF_q. Motivated by the excellent results in NLP, BERT TF follows the concept of the well known Transformer model BERT [41], removing the decoder

from the main architecture, but this approach did not give good results. TF_q instead was developed to make comparisons against models which do not follow the single trajectory deterministic approach. In TF_q, people motion is quantized in 1000 speed clusters bins, so that the position is encoded by a 1000 dimensional one-hot encoded vector. With this formulation, the problem is not anymore in a regression form, but in a classification form, since the goal is to classify the correct bin at each step. The advantage is that in this way it is possible to sample distributions of trajectories from the predicted outputs, thus to sample multiple trajectories. Also this model resulted in competitive results against other stochastic state-of-the-art models.

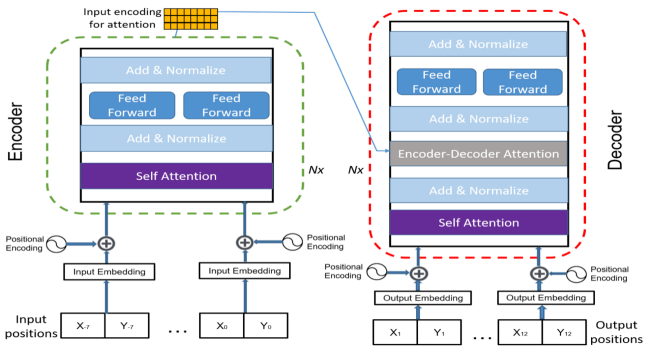


Figure 4.5: Transformer TF architecture, from [7].

4.2.2 AgentFormer

AgentFormer [8] is a Transformed based stochastic model currently state-of-the-art for trajectory forecasting using the non deterministic protocol. Previous methods usually modelled the *time* dimension (where each agent state - position or velocity - influences the future agent states) and the *social* dimension (behaviour of an agent influences the others) separately, for example considering only one of the two or considering first one dimension and then the other dimension on the computed previous features. AgentFormer models the two dimensions together by using a novel attention structure, allowing each agent in the scene to directly attend to information of the past of the other agents. The trajectories in input are flattened across time and agents:

- The *time* information is preserved by a time encoder which timestamps the agent states.
- The *agent* information is preserved through the use of the novel attention mechanism called *agent-aware attention*. This type of attention changes with

respect to the original Transformer one, because it generates two different sets of keys and queries, one set used for intra-agent attention (attention to itself) and the other used for inter-agent attention (attention to the other agents).

The stochasticity is achieved through the use of a CVAE framework, where the future trajectory is conditioned on the context (semantic maps and past trajectory). The latent code z represents the latent intent of each agent and it is incorporated at the beginning of the decoder allowing each agent latent code to influence the other agents. In Figure 4.6 the overall AgentFormer architecture.

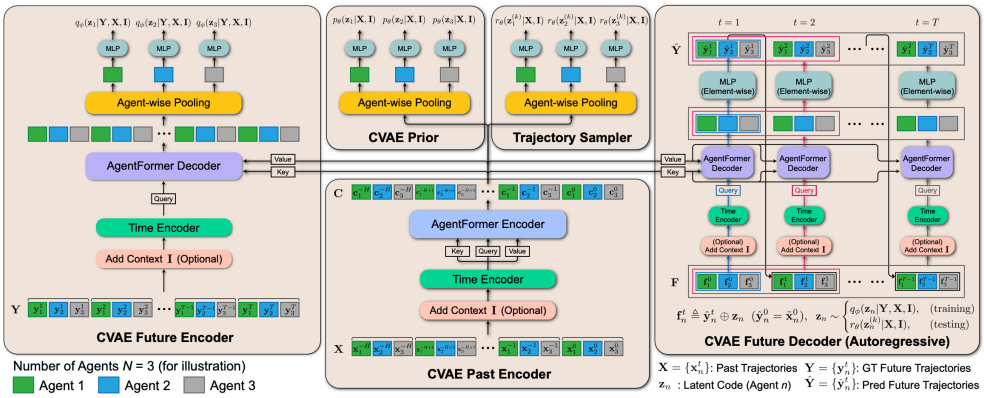


Figure 4.6: AgentFormer architecture, from [8].

- The *past encoder* encodes the past trajectory using an AgentFormer encoder.
- The *prior* module outputs the parameters of the prior distribution.
- The *future encoder* module outputs the parameter of the approximate posterior on which the Kullback-Leibler divergence is computed.
- The *future decoder* autoregressively outputs the trajectories given the latent codes and past trajectories.
- The *trajectory sampler* module adopts a diversity sampling technique [42] to generate latent codes which generate diverse and plausible trajectories. This module is trained once the CVAE is already trained, freezing its parameters.

During training, the latent code is generated by the approximate posterior q_ϕ, while in testing, without the *trajectory sampler* module, the latent code is obtained through the *prior* module. The authors proposed to add the *trajectory sampler* in order to generate useful and diverse latent codes, since many times similar latent codes lead to similar trajectories. This module, trained on the freezed model, makes

it possible to generate a set of latent codes which cover also possible minor modes, that random sampling would hardly generate.

Chapter 5

Experiments

5.1 Introduction

In this chapter, the experiments and methodology used are presented. Firstly, the datasets and metrics used in the experiments proposed are introduced in Sections 5.2 and 5.3 respectively.
Then, in section 5.4 the main model from literature used in the experiments is presented and several experiments are performed on it in order to understand its behaviour and improve its performances changing the loss or adding techniques of data augmentation.
Lastly, an extensive effort is put in Section 5.5, where a Transformer based GAN is developed using the previous findings, in order to model the stochasticity and multimodality of human behaviour. In this section the generative model applied to the task, its training procedure and a novel sampling technique to generate diverse and plausible multiple predictions which cover the output space are introduced.
Every section presents the corresponding results with the comparison with previous or literature results.

5.2 Datasets

To efficiently train neural networks it is fundamental to have enough data, and it must be of a good quality too, because the model will learn to reproduce and infer based on the data it has seen during training.
Pedestrian trajectory forecasting datasets can use two different types of coordinates:

- Image coordinates: the coordinates are represented by the pixel position on the image. This type of coordinates is specific to the input image and do not represent real world distances (a closer or further camera would have two different results).

- World coordinates: the coordinates are represented by the relative distance in meters to an arbitrary chosen point (origin) of the world. This type of coordinates is invariant to the position of the camera, as long as the chosen point of origin is the same. The only drawback is that this type of coordinates is more difficult to obtain, first the image coordinates are extracted from a source and then they are transformed thanks to a holographic matrix to the world version. If the holographic matrix of the scene is not available, it is not possible to convert image coordinates to the world version and viceversa.

In this work, the used datasets ETH and UCY detailed below, are in world coordinates. They have been chosen for being the most used in literature in the field of pedestrian trajectory forecasting.

5.2.1 ETH and UCY

ETH [43] and UCY [44] are two of the most used datasets for benchmarks in pedestrian trajectory forecasting. They both are in world coordinates and, even if not captured together or by the same team, they are used together for their similar characteristics. Each of the two datasets is subdivided in smaller datasets.
ETH comprises two scenes, ETH and Hotel, where the coordinates are extracted from videos taken from a fixed camera bird's eye position. The two scenes together count 750 pedestrian trajectories annotated at 2.5 Hz.
The same characteristics are shared with UCY (bird's eye view, fixed camera and 2.5 frame rate), which consists in 3 sub-datasets: Univ, Zara1 and Zara2, counting 786 total trajectories.
Since the two datasets are almost always used together, we can consider them as a single dataset named ETH-UCY from now on. This composed dataset has 5 scenes in total (ETH, Hotel, Univ, Zara1, Zara2) that are usually evaluated using a leave-one-out strategy. This means that in training, 4 scenes of the 5 are used as training sets while the left out one is used for testing. This procedure is repeated once for each scene, meaning that to evaluate a single model on all 5 scenes, the model must be re-trained 5 times.
The datasets and train/val/test splits used for training the models of this master thesis are the ones available at the GitHub repository of the Transformer TF model [7].
In Figure 5.1 two snapshots of the original videos of ETH-UCY can be observed, the agents' coordinates were then extracted from these videos.

Figure 5.1: Frames of Zara scene (left) and Univ scene (right). Image coordinates are extracted from video and transformed to world coordinates.

5.3 Metrics

To evaluate a model performance, it is important to have a measure of goodness of the solution proposed. This measure gives a numerical value on the performance of the model, allowing to compare them.
In the case of pedestrian trajectory forecasting, the two most used metrics in literature are the average displacement error and the final displacement error explained in the next subsections.

5.3.1 Average Displacement Error (ADE)

The Average Displacement Error (ADE) is one of the two most important metrics used to assess model performance in pedestrian trajectory forecasting. As the name suggests, it computes the overall distance error between ground truth points and predicted points. The error is the Euclidean distance between the true and predicted positions averaged over all pedestrians and number of predicted timesteps. If we consider n the number of pedestrians, T_{obs} the observed timesteps, T_{pred} the predicted timesteps and $\hat{Y}_t^i$, Y_t^i the predicted and true positions respectively of agent i at timestep t, we can write the ADE as:

$$ADE = \frac{1}{n(T_{pred} - T_{obs})} \sum_{i=1}^{n} \sum_{t=T_{obs+1}}^{T_{pred}} ||\hat{Y}_t^i - Y_t^i|| \tag{5.1}$$

The measure is the average distance of the predicted positions to the ground truth ones, which gives an average distance in meters.

A variation of the ADE is the TopK-ADE which is typically used in models that do not use a single trajectory deterministic protocol. Since these models output

multiple trajectories, it would not have much sense to average the error on all the predicted trajectories because these models often try to capture the multimodality aspect of the trajectory forecasting problem. This aspect is very difficult to model because the ground truth trajectory is only one, so we need not to penalize too much models which make multiple different plausible predictions. In order to do so, TopK-ADE is used, where the model makes K predictions and the ADE is computed only on the best predicted trajectory out of the K. This type of metric is sort of a lower bound because it is obvious that forecasting multiple trajectories lowers the value of the ADE metric, while at the same time it does not control the plausibility of the other trajectories.

5.3.2 Final Displacement Error (FDE)

The Final Displacement Error (FDE) is the other very important metric used in trajectory prediction. It consists in computing the distance error for all pedestrians, but only on the last timestep of the trajectory. In practice, it computes the distance between the final position of the ground truth trajectory and the final position of the predicted trajectory, and it is averaged between all pedestrians. Using the same notation used for ADE, we can formally define FDE as:

$$FDE = \frac{1}{n} \sum_{i=1}^{n} ||\hat{Y}^i_{T_{pred}} - Y^i_{T_{pred}}|| \tag{5.2}$$

This metric gives us an average distance in meters since we are dealing with world coordinates datasets.

Also in this case, it exists the version for multiple trajectory models called TopK-FDE and it is equivalent to TopK-ADE. Instead of averaging the FDE across the K predicted trajectories, which would not make much sense since these models usually want to learn multimodal distributions, the FDE is computed for all predicted trajectories, and the best one out of the K is chosen. Also in this case it is sort of a lower bound to the standard FDE, since predicting more trajectories makes it more likely to predict at least one which is closer to ground truth.

An illustrated example of the two metrics is available in Figure 5.2.

5.4 Chosen model

The main model used in the experiments is the Transformer TF [7], already described in the literature review in Chapter 4. This model was chosen because of its trade-off between simplicity and performance. With respect to other more complex models, this is a standard application of the Vanilla Transformer by Vaswani et al. [6] to the trajectory forecasting task. For this reason multiple experiments could

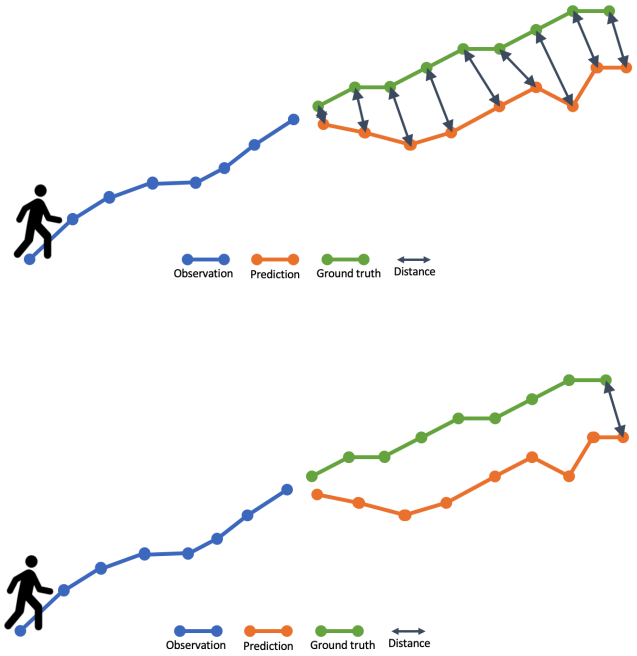


Figure 5.2: ADE metric in the top, the distance between true and predicted position is averaged across timesteps. FDE metric in the bottom, only the last timestep is used to compute the distance.

be done on it and a new stochastic approach can be used to train the model on multiple trajectories. Furthermore, the model does not use any information about the map nor the social part and its performance is not much worse than the current state-of-the-art models which make use of this information. This can indicate that both social and map information are useful but up to a certain extent, and similar results can be obtained without them. The original results of the Transformer TF are available in Table 5.1.

In the next subsections, some introductory experiments were done to the original deterministic version of the model, starting from modifying the embedding dimension of the input coordinates, trying a new loss, changing the coordinates encoding and trying some data augmentation techniques.

5.4.1 Input embeddings dimension

The standard transformer uses an embedding layer to transforms the input to a given size and then performs many layers of the encoder/decoder to the embedded

Dataset	ADE	FDE
ETH	1.03	2.10
Hotel	0.36	0.71
UCY	0.53	1.32
Zara1	0.44	1.00
Zara2	0.34	0.76
avg	0.54	1.17

Table 5.1: Results of the original Transformer TF following the single trajectory deterministic protocol. Results are reported from original paper [7]. The models are trained on 4 datasets and tested on the remaining one, following the Leave-One-Out approach.

input where positional encoding has been applied.
In Transformer TF, the mechanism is the same and the input coordinates (2 dimensional points which features are the x-axis relative position and the y-axis relative position to the previous timestep) are transformed to a d dimensional vector. The authors of Transformer TF propose to use 512 dimensional embeddings, in other words, to learn a projection of the 2D coordinate to a 512 dimensional space. This approach could be not optimal and it could be possible to use much lower dimensions to project the data to. In fact, the embedding dimension is one of the most important hyperparameters which affects the number of learnable parameters of a transformer model, since the embeddings are used in the multihead attention and having a much lower embedding dimension reduces the number of operations needed for the computation of the attention. The number of learnable parameters in Transformer TF with all the different embedding dimensions is available in Table 5.2. It is possible to notice how halving the embedding size reduces the parameters by more than half. So, decreasing this dimension brings multiple advantages, it makes the model smaller with less learnable parameters, and consequently it increases the speed of training, and it also makes the model less prone to overfitting. In fact, we can see an overfitting trend reported in Figure 5.3, where we can see the training loss improving for all embedding dimensions while the performance on the validation set decreases during training for higher dimensional embeddings.

Implementation details

The training of the models was done in 120 epochs, batch size of 64 and using the same hyperparameters settings of the original Transformer TF (6 encoder and decoder blocks, Transformer hidden linear layers of 2048 dimension and multihead

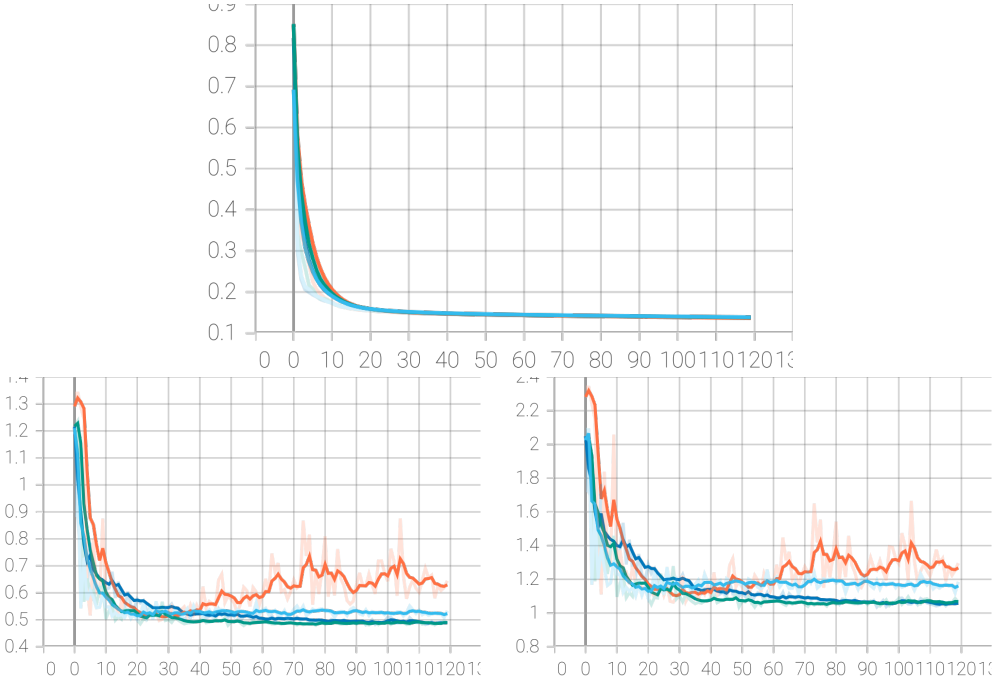


Figure 5.3: Training curves for the Zara2 dataset with diverse embedding dimensions. On the top the training loss, on the bottom the ADE and FDE, left and right respectively, computed on the validation set. Orange is 512 emb., green is 256 emb., light blue 128 emb. and blue is 64 emb.

Embedding dimension	Learnable parameters
512 (Original)	44.145.667
256	17.366.531
128	7.515.907
64	3.475.331

Table 5.2: Number of learnable parameters in Transformer TF while varying the embedding dimension. By decreasing embedding size we speed up the training and have a less complex model, while keeping the performances similar for this task.

attention with 8 heads). It is important to notice that the embedding dimension has a significant impact also on the optimizer used for training the transformer. The comparison between model performances has been done with the original optimizer of the transformer architecture which is the Noam Optimizer. Noam Optimizer is a wrapper which uses the Adam Optimizer with a learning rate scheduler that depends on some parameters, such as embedding dimension and warmup steps.

The formula for the learning rate is:

$$lrate = d_{model}^{-0.5} \times \min(step\_num^{-0.5}, step\_num \times warmup\_steps^{-1.5}) \quad (5.3)$$

As we can see, the embedding dimension d_{model} influences very much the curve. An example of the learning rate scheduling for the models when the embedding dimension change is illustrated in Figure 5.4. For the number of warmup steps, each model was trained in order to reach the maximum at the 10th epoch.

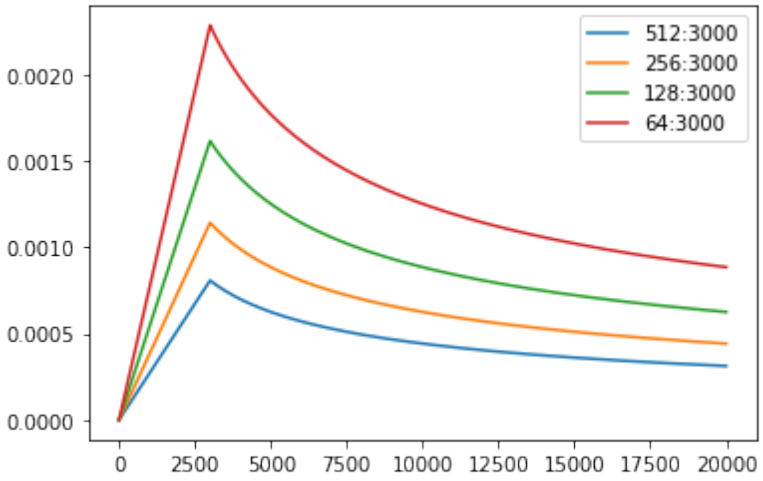


Figure 5.4: Learning rate scheduler for different embedding dimensions. The warmup steps are 3000.

Results

The effects of changing the embedding dimension can be observed in the results Table 5.3. Since the embedding dimension reduction did not have a much negative impact on the average results, the embedding dimension chosen was 64. This embedding dimension is also used for all other experiments in the thesis, having a good trade-off between performance and model complexity. These considerations are assumed to hold also in the case of the stochastic model proposed in the second part of the experiments. They are assumed to hold because the functioning of the model is of the same form (it considers each trajectory singularly and makes one single prediction at each forward pass) and because of time constraints, not allowing each model to be trained by optimizing for every hyperparameter. For these reasons, the best results are chosen at each experiment and assumed to propagate to future ones.

Dataset	512 emb. (Original TF)	256 emb.	128 emb.	64 emb.
ETH	1.03/2.10	**1.01**/2.09	1.02/**2.05**	1.06/2.16
Hotel	**0.36/0.71**	0.46/0.99	0.43/0.93	0.44/0.93
UCY	**0.53**/1.32	0.56/**1.22**	0.56/1.23	0.57/1.22
Zara1	0.44/1.00	0.42/0.93	0.43/0.98	**0.41/0.92**
Zara2	0.34/0.76	**0.32/0.70**	0.33/0.76	0.33/0.72
avg	**0.54/1.17**	0.55/1.19	0.55/1.19	0.56/1.19

Table 5.3: ADE/FDE metrics on ETH-UCY datasets while varying the embedding dimension. 512 originally used in TF.

5.4.2 Loss

As for every deep learning model, the training procedure needs a loss which measures the error between model predictions and ground truth, and makes it possible to compute the gradients to update the model parameters.
The loss used in the original Transformer TF implementation is the Pairwise Distance, which computes the pairwise distance between two vectors v and u using the p-norm. The vectors u and v are 2 dimensional containing the (x, y) coordinates of the agent: one containing the ground truth coordinates, the other containing the predicted ones. We define the difference vector of the two as:

$$d = v - u = (v_x, v_y) - (u_x, u_y) \tag{5.4}$$

where each of the two components is the element-wise difference of the two former vectors u and v. Now we can formally define the p-norm of the vector d as:

$$||d||_p = \left(\sum_{i=1}^{2} |d_i|^p \right)^{1/p} \tag{5.5}$$

and the loss for a single trajectory is the average of the p-norm evaluated at every predicted timestep:

$$\mathcal{L} = \frac{\sum_{j=T_{obs}}^{T_{pred}} (||d||_p)_j}{T_{pred} - T_{obs}} \tag{5.6}$$

that is, for each predicted trajectory, the loss is technically the Average Displacement Error if the value of p is 2. With that value the p-norm becomes the Euclidean distance which is how the Average Displacement Error is computed.
In this preliminary experiment phase, two types of p-norm were confronted, the 2-norm and the 1-norm. The reason for experimenting with the latter is that for small discrepancies the 1-norm has higher values which could in theory benefit the model pushing the predictions closer to ground truth.

Implementation details

The hyperparameters are set the same as for previous section, 120 epochs for the training and a 64 dimensional embedding dimension. The optimizer is Noam Optimizer with the learning rate scheduler explained before, as suggested by literature best practices to train a Transformer model.

Results

In Table 5.4 the comparison of the results using the two losses. In the experiments, the best performing loss was the one using the 2-norm. This can be explained by the fact that the 2-norm loss actually is the Average Displacement Error, which is a metric used for testing the trained models. So, the model learns the best possible parameters values to minimize the loss that is also used as a metric for testing, making train and test time more similar and consistent. The performance difference is not very high but the 2-norm consistently outperforms the 1-norm counterpart. For this reason, for further experiments, the 2-norm was always used.

Dataset	1-norm	2-norm (Original)
ETH	**1.04/2.12**	1.06/2.16
Hotel	0.51/1.08	**0.44/0.93**
UCY	**0.57/1.22**	**0.57/1.22**
Zara1	0.44/0.98	**0.41/0.92**
Zara2	**0.33**/0.73	**0.33/0.72**
avg	0.58/1.23	**0.56/1.19**

Table 5.4: ADE/FDE results using a 2-norm loss and a 1-norm loss. 2-norm originally used in TF.

5.4.3 Coordinates encoding

The Transformer TF receives only the coordinates of the agents as input, but there are various ways to encode this data, as suggested also by [45]. In fact, as is shown in this section, the encoding type can significantly impact the model performance and its capability to generalize.
Given the problem and the type of data, four possible encoding methods were investigated:

- *Raw coordinates*: where the coordinates are not manipulated in any way, and fed to the model as originally in the dataset. This is the simplest approach which theoretically would perform the worst since each dataset of the five

represents a different scene and has a different position for the origin of the plane. Given the testing criteria, where a model is trained on four datasets and tested on the remaining one, a model trained with these coordinates should not be able to generalize to the unseen dataset.

- *Relative coordinates*: in this case the coordinates are represented by the relative displacement in the x and y axis with respect to the previous timestep. This encoding is the one originally used in Transformer TF and, intuitively, it is preferable with respect to raw coordinates since in this way the model cannot associate specific parts of the environment with specific motion behaviours. To reconstruct the original trajectory, the raw coordinates of the first timestep are needed and the x and y displacements are cumulatively added to the initial point.
- *Coordinates centered in the last observed point*: using this kind of encoding the origin of the plane is centered on the last observed timestep T_{obs-1}.
- *Coordinates centered in the first observed point*: this type of encoding is similar to the previous one, but with the difference that the translation of the coordinate plane is not on the last observed timestep, but on the first one, making the first point in the trajectory sequence with coordinates [0,0].

An example of different coordinates encodings is available in Figure 5.5.

Implementation details

Same hyperparameters of previous sections, embedding dimension of 64, 2-norm loss and training of 120 epochs.

Results

Overall, the relative coordinates are the best performing ones on all datasets as can be seen in Table 5.5 where the results are shown. Interestingly, they were not only the best performing, but also the fastest converging (Figure 5.6). The other methods other than the raw coordinates could have reached similar results, but in much more training time as shown in the Figure 5.6. This behaviour could be due to the fact that with relative coordinates the only focus is in predicting the displacement with respect to just the previous timestep, while in the other methods, as the timesteps increase, the magnitude of the prediction increases and the output possibilities are more. Raw coordinates poorly performed because they are a type of coordinate representation which is not scene invariant and this causes the model to learn scene patterns that, by training the models with a Leave-One-Out approach, is going to negatively affect performance.

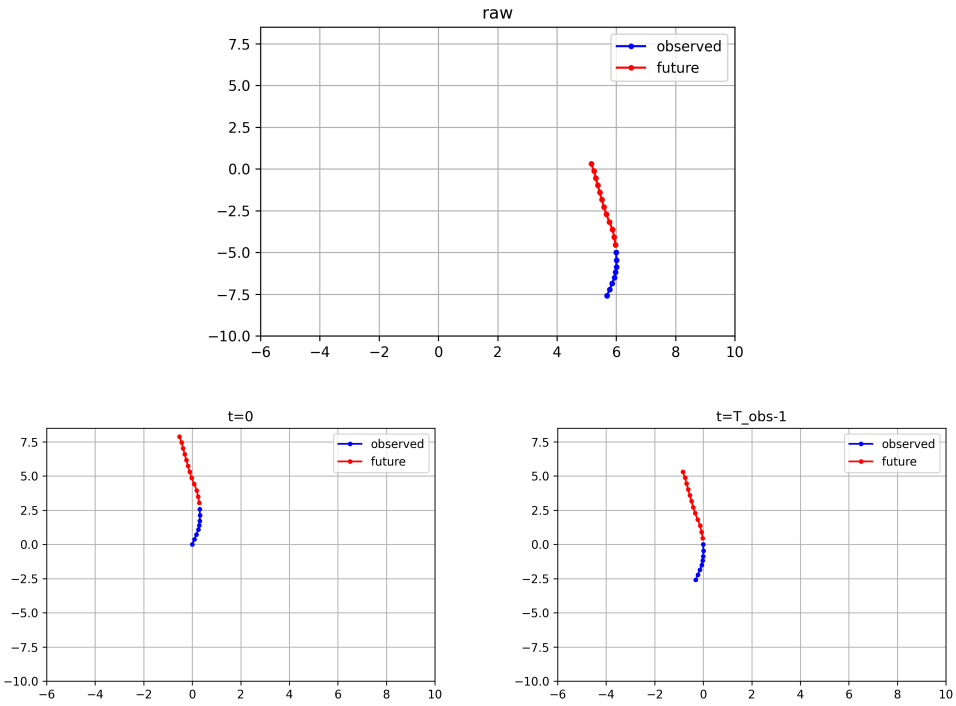


Figure 5.5: Representation of different coordinates encodings. On the top raw coordinates, bottom left coordinates centered in the first timestep, bottom right coordinates centered on the last observed timestep.

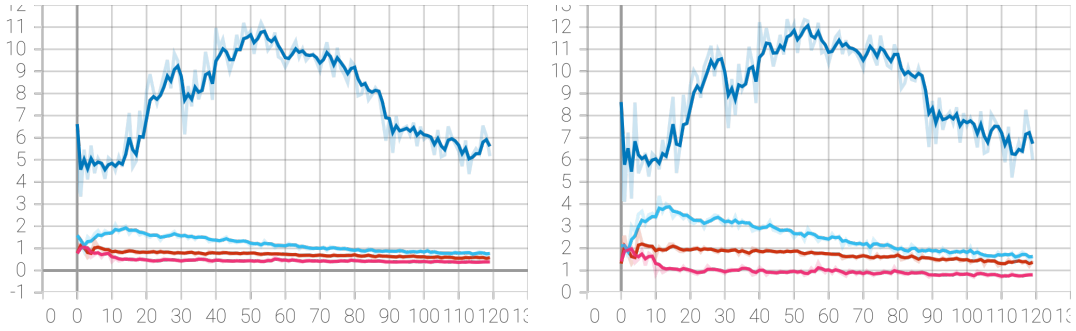


Figure 5.6: Training curve of a model tested on the Hotel dataset. On the left the ADE and on the right the FDE. We can observe the performance differences between raw coordinates (blue), coordinates centered in the first point (light blue), coordinates centered in T_{obs-1} (red) and relative coordinates (magenta).

Dataset	Raw	Relative (Original)	$t = 0$	$t = T_{obs-1}$
ETH	1.52/2.68	1.06/2.16	1.24/2.50	**1.04/2.05**
Hotel	4.36/5.20	**0.44/0.93**	0.68/1.39	0.48/1.16
UCY	2.23/4.21	**0.57/1.22**	1.22/2.60	0.68/1.55
Zara1	0.49/0.96	**0.41/0.92**	0.80/1.95	0.55/1.50
Zara2	0.54/1.08	**0.32/0.72**	0.68/1.51	0.41/1.05
avg	1.83/2.83	**0.56/1.19**	0.92/1.99	0.63/1.46

Table 5.5: Results of the models using the different coordinates encodings. Relative encoding is the one originally used in TF.

5.4.4 Data augmentation

Another set of experiments done was the one regarding data augmentation. Data augmentation is a technique widely used in deep learning that consists in producing more samples of the training data. It is used because in order to learn the patterns in the data, models usually need very high numbers of samples and making new ones from the already available ones is a useful technique. If intelligently used, data augmentation can benefit the model performance even by large margins. Moreover, it also reduces the risk of overfitting since it becomes more difficult for the model to learn the exact training data patterns.
In the original implementation of Transformer TF no data augmentation was used. The ETH-UCY datasets of the experiments are not enormous datasets, so, following also prior work which uses mostly random rotations of the scene [5, 8, 46] it was decided to apply different data augmentations to see the effects on the results. Each one of the data augmentations proposed is applied on the single trajectory and not on the whole scene selected. So, for example, if a scene comprises three agents and consequently three trajectories, each one of the trajectories is singularly selected and transformed using one or more of the four data augmentation techniques investigated:

- *Rotation*: the rotation data augmentation rotates randomly each trajectory in the training set, every time it is selected. In this way, the patterns learned are not specific to any kind of scene or environment and the model is forced to learn the human motion behaviour instead of the structure of the scene. In our leave one out testing approach, this should intuitively help the model generalize to unseen environments.

- *Mirroring*: mirroring refers to flipping the trajectory with respect to the x or y axes. Every time a trajectory is selected from the training set, a x, y, both axes or no mirroring is applied with a 1/4 probability.

- *Noise on the whole trajectory*: adding noise to the trajectory could help the

model to be more robust to noisy inputs with little errors as it can be from sensors inputs. In this setting the noise sampled from a Normal distribution $\mathcal{N}(0,0.1)$ is added on the whole trajectory, both on the observed timesteps and the ground truth points to predict.

- *Noise only in the observed trajectory*: another possible setting for the noise data augmentation is to add noise only on the observed timesteps and leave the ground truth points unchanged.

An example of the random rotation data augmentation can be seen in Figure 5.7.

Results

The results of the experiments are available in Table 5.6. We can observe how the rotation augmentation is the most helpful in terms of performance, while adding also mirroring does not improve the results obtained with random rotations, keeping the performance the same. Lastly, we see that adding noise to the trajectories is actually detrimental on average. Adding noise to the whole trajectory worsens the results in all the datasets while adding it only to the observed timesteps worsens the results on almost all datasets with the exception of Hotel. It is possible to conclude that the more timesteps noise is added to, the more performance drops and this data augmentation is not useful to learn motion behaviour with more robustness.
Validated by these results, the only data augmentation used in further experiments is the random rotation data augmentation.

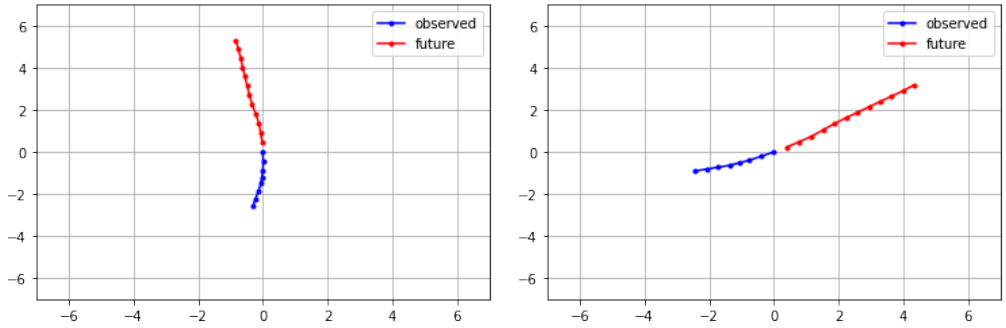


Figure 5.7: Example of original trajectory (left) and rotated trajectory (right)

Dataset	Original TF	Rot	Rot + Mir	Rot + Mir + Noise	Rot + Mir + Noise_src
ETH	1.03/2.10	1.00/**2.02**	1.03/2.12	1.00/2.05	**0.99/2.02**
Hotel	0.36/0.71	0.33/0.65	0.35/0.70	0.40/0.87	**0.32/0.63**
UCY	0.53/1.32	0.55/1.19	**0.54/1.15**	0.67/1.35	0.63/1.28
Zara1	0.44/1.00	0.44/0.99	**0.43/0.96**	0.74/1.63	0.48/1.02
Zara2	**0.34**/0.76	**0.34/0.75**	**0.34**/0.76	0.49/1.01	0.38/0.80
avg	0.54/1.17	**0.53/1.12**	0.54/1.14	0.66/1.38	0.56/1.15

Table 5.6: Results of the experiments on data augmentation techniques.

5.5 Stochastic model

In this part of the thesis the multimodal and stochastic aspects of human motion have been attempted to be modelled. As previously stated, pedestrian motion is by nature not deterministic and it is not straightforward to model this aspect since the ground truth future given for each trajectory is only one among multiple (possibly infinite) plausible trajectories. For example, if a pedestrian is at a crossroads, the possible trajectories are multiple and neither of them is actually wrong, it depends on where the agent needs to be going to. The same happens if a pedestrian faces another pedestrian that is coming from the opposite direction, obviously the two agents need to take into account the possible collision between them and correct their trajectories to avoid it, but this could happen in multiple ways which in most cases are not wrong in terms of *social etiquette*. The problem in modelling these scenarios is that deep learning models need a ground truth trajectory to try to understand the underlying human motion distribution, and this given trajectory is the one which actually occurred in the training examples, leaving outside all the other possible motions which didn't occur.
Obviously, if the agent intent was fixed and given, various cases in which multiple futures are plausible would become much easier to model (for example the crossroads example), but this is not the case since the arrival point is not given in any of the datasets.
Another challenge faced when dealing with this task, which is the consequence of previous problems, is that there is no explicit way to tell whether a predicted trajectory is right or wrong. There are some motions that are surely wrong (e.g. walking inside a wall, collide with other agents, etc.), but for many other occasions there is no right way to tell if a prediction is good or bad.

These are the main challenges that comes when approaching this task. Many approaches in literature tried to model multimodality by using generative models [4, 8, 5]. Usually, a multimodal model performance is evaluated by using the TopK-ADE and TopK-FDE which are explained in the metrics Section 5.3, they

consist in predicting multiple trajectories and select the best one in terms of regular ADE/FDE.

In the next sections a new model, Transformer GAN, developed in this thesis is proposed, a modified sampling method to improve multimodal model performance is applied to the model and, lastly, a comparison with results obtained by models present in literature is shown.

5.5.1 Proposal: Transformer GAN

The model proposed is a Generative Adversarial Network with the core generator made by almost the same architecture as Transformer TF. A generative model approach was decided to be used because of the inherent stochasticity present in these kind of models, which can be helpful in modelling the multimodality of the trajectory forecasting task. Moreover, the GAN approach used is inspired by Social GAN [4] where the stochasticity is modelled by using a latent vector z sampled from a Normal distribution, every time a trajectory is predicted. This means that for the same pedestrian, multiple trajectories can be predicted by sampling multiple latent codes. Motivated by the good results of that model at that time and by the fact that no other model used a GAN framework, others used mostly CVAEs generative models and especially mostly not with Transformers yet, it was decided to see how this kind of model could be developed without LSTMs and how it would have performed. The Transformer TF architecture was chosen for its great sequence modelling performance, as seen in previous Section for the deterministic task. The only drawback was that the Transformer TF is not capable of multiple predictions and, in order to overcome its deterministic nature, a latent vector was introduced and the model was incorporated inside a GAN framework.
The architecture of the proposed model is available in Figure 5.8.

Generator

In a generative adversarial network, the generator is the part of the network devoted to generating the samples. In this application, the generator is very similar to the Transformer TF model, with the addition of a latent vector embedding to the inputs of the decoder. The inputs of the decoder are firstly passed through the input embedding which projects the coordinates to fixed length d_{model} dimensional vectors, then these embeddings are concatenated to a latent vector z of dimension d_z and passed through a linear layer to project the concatenated vectors to d_{model} dimension. In this way the stochasticity is introduced in the model. Positional encoding is then applied and the result of previous steps is then passed to the Transformer TF decoder which produces in output the predicted trajectory conditioned both on past motion and latent code.

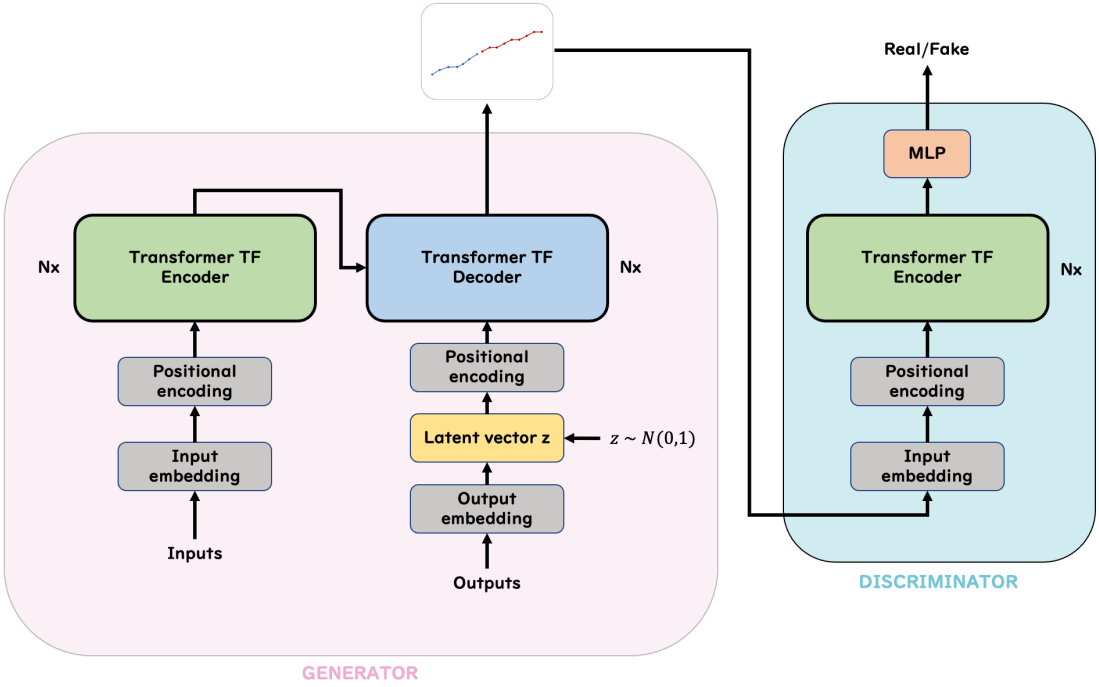


Figure 5.8: Transformer GAN architecture.

Discriminator

The discriminator is the part of the network working in contrast to the generator and its job is to differentiate between samples that are real or generated by the generator. In this case the discriminator is composed by a separate Transformer TF encoder without the need for a decoder. It takes in input the whole trajectory and classifies it as real or fake: real if it is the ground truth trajectory of the training set, fake if it is the trajectory predicted by the generator. In order to output this information, the encoder outputs the encoded trajectory which is then passed through an MLP with only one output neuron that indicates if a sample is real or fake.

Training

The training of a GAN is different than training a standard model, because in this case the models to train jointly are two, the generator and the discriminator. They compete with each other with the generator trying to fool the discriminator and the discriminator trying to understand the tricks of the generator.
The training procedure iteratively consists in two steps, the generator step and the discriminator step. During these two steps only a model at a time is trained while

the other is frozen. In practice the two steps are described as follows:

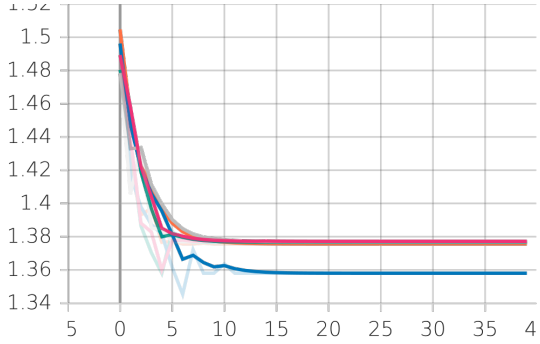
- *Discriminator step*: where only the discriminator is trained. A batch of data is used in this step and the generator produces a prediction for the batch. Then, both the predicted and true trajectories are fed to the discriminator which outputs the probability scores of being a real sample (1 if the sample is real, 0 if it is fake). Once these scores are obtained, the real samples loss and the predicted samples loss are computed and backpropagated on the discriminator, trying to make it more accurate in distinguishing between real or generated trajectories. The loss used in this step (Figure 5.9a) is the binary cross entropy loss, defined as:

$$BCE = -\frac{1}{N}\sum y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i) \tag{5.7}$$

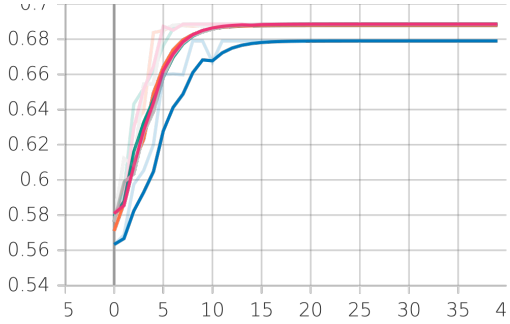
  where y_i and $\hat{y}_i$ are the real score and predicted score of the i^{th} sample.

- *Generator step*: in this step only the parameters of the generator are updated. A new batch of data is used for this step where the generator makes its trajectory predictions. These predictions are then fed to the discriminator, but this time we tell the loss that these trajectories are instead real trajectories and that it should classify them in that way. This passage is very important because it is the way that the generator learns to output trajectories which confuse the discriminator. If, for example, the discriminator easily discerns that the trajectories are fake, it means that the generator is not doing a good job and it should change its output to make it more difficult for the discriminator.
  In this step, two losses are used: a binary cross entropy loss on the output of the discriminator (Figure 5.9b), and the pairwise distance loss (Figure 5.9c) already used in the model Transformer TF to also force the generator to produce samples that not only fool the discriminator, but that also cover ground truth.

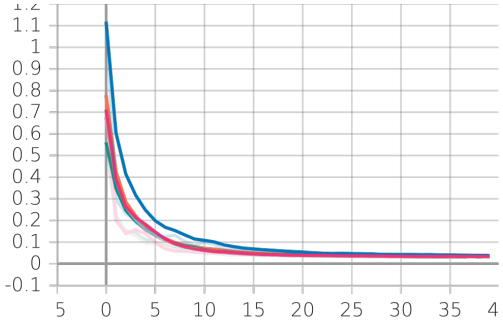
The learning curves for each model are observable in Figure 5.9.



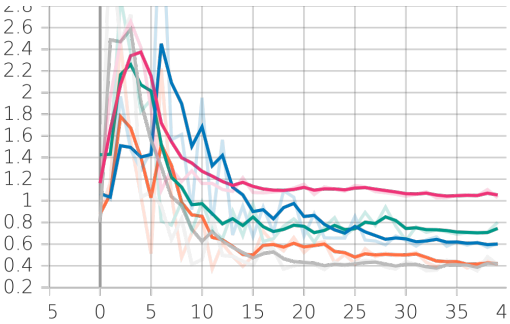
(a) Discriminator loss



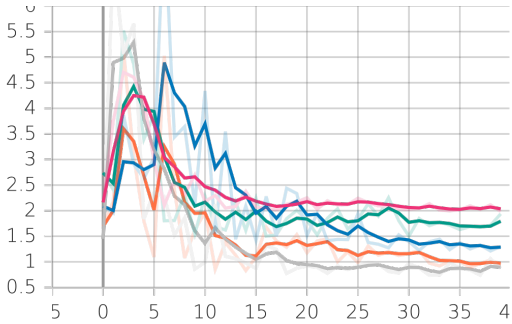
(b) Generator BCE loss



(c) Generator L2 loss



(d) Test ADE



(e) Test FDE

Figure 5.9: Learning curves for the GAN. ETH magenta, Hotel grey, UCY blue, Zara1 green and Zara2 orange.

Implementation details

The findings from previous sections were used in training the models. The embedding dimension used is 64, the coordinates encoding used is the relative one and the only data augmentation used is rotation. The models were trained for 40 epochs and generator and discriminator, being two Transformer models, used

two separate optimizers which are two Noam Optimizers with the learning rate peak at the 10^{th} epoch. The number of discriminator steps and generator steps at each iteration was one for both, noisy labels were used for the discriminator such that real labels of 1.0 were replaced with a random number between 0.7 and 1.2 and fake labels of 0.0 were replaced with a random number between 0.0 and 0.3 following literature best practices as suggested in [47].

5.5.2 Sampling method

In this section the sampling method used is described and then the results obtained are shown. As previously stated, training the Transformer GAN leads to the model learning a unimodal trajectory distribution and consequently almost no differentiation in the predicted trajectories when sampling various latent vectors z. For this reason, once introduced the latent vector model Transformer GAN, we now need to differentiate the multiple trajectories predicted. A way to do so is by sampling the k latent vectors z in such a way that the outputs generated from these samples are different and cover various modes. The method applied is conceptually very similar to a method present in literature called DLow by the authors [42], but it presents a major difference in the training procedure. First, the original method is introduced.

DLow

Diversifying Latent Flows (DLow) [42] is a novel sampling method for pre-trained generative models. Its objective is to generate diverse samples once the model has already been trained. Usually, these models, as Transformer GAN, draw a set of independent Gaussian latent codes which are then decoded in the predictions. But there are some drawbacks with this strategy, for example the independent sampling cannot force the generated samples to be diverse and the generated predictions usually are very similar and correspond to the major mode.
To overcome these issues, DLow consists in a sampling method that samples just once from a single random variable and then maps the sampled latent code to a set of k correlated different codes. The parameters which map the original sample to the final k are learned after the generative model is trained in such a way to maximize the diversity among the k predictions while keeping the generated latent codes likely under a prior.
In standard generative models, the process of generating a sample follows:

$$z \sim p(z) \tag{5.8}$$

$$x = G_\theta(z, c) \tag{5.9}$$

where z is the latent code sampled from its prior distribution $p(z)$, usually a Gaussian, and G is the generator model parametrized by θ that produces the

sample x based on the latent code z and the conditioning c, which could be the past observed trajectory for example.

Differently, DLow introduces a new random variable ϵ and generates the k latent codes dependently on ϵ as:

$$\epsilon \sim p(\epsilon) \tag{5.10}$$
$$z_i = \mathcal{T}_{\psi_i}(\epsilon) \quad 1 \leq i \leq k \tag{5.11}$$
$$x_i = G_\theta(z_i, c) \quad 1 \leq i \leq k \tag{5.12}$$

where $p(\epsilon)$ is a Gaussian prior and $\mathcal{T}_{\psi_i}$ are the latent mapping functions with parameters ψ_i that map ϵ to the k latent codes z_i.
The k latent codes are correlated as they are uniquely determined by ϵ. The objective of DLow instead is defined as:

$$\mathcal{L}_{DLow} = \mathcal{L}_{diversity} + \beta \mathcal{L}_{KL} \tag{5.13}$$

where $\mathcal{L}_{KL}$ is the KL divergence between the k generated latent codes distributions and the Gaussian prior $p(z)$ and $\mathcal{L}_{diversity}$ is a diversity promoting loss between the generated samples x, which formulation is task dependant.
The latent mappings $\mathcal{T}_{\psi_i}$ transforms the latent distribution $p(\epsilon)$ to the marginal latent distribution $r_{\psi_i}(z_i, c)$, so to be that the case, $\mathcal{T}_{\psi_i}$ should be also conditioned on c. Since the latent codes z_i should also follow a Gaussian distribution, the authors designed $\mathcal{T}_{\psi_i}$ to be an invertible affine transformation of ϵ:

$$\mathcal{T}_{\psi_i}(\epsilon) = A_i(c)\epsilon + b_i(c) \tag{5.14}$$

where the mapping parameters are $\psi_i = \{A_i(c), b_i(c)\}$ with $A_i(c)$ a $n_z \times n_z$ matrix and $b_i(c)$ a n_z dimensional vector. The parameters are estimated through a network Q. Since the transformation transforms the prior distribution $p(\epsilon)$ to another Gaussian distribution (by rotating and translating the plane), the KL divergence can be computed analytically which is a great benefit of this method.
An overview of the architecture is available in Figure 5.10.

Proposed method

The proposed sampling method is heavily inspired by DLow and applies the same process, but the difference is that in our training of the model, the generator parameters are not kept fixed. We assume to have an already trained instance of the Transformer GAN generator (one instance for every different dataset) and apply the method to it. The reason behind not keeping the weights fixed is that in this way, after the model has already been trained and it has learned the most likely trajectory distribution in a unimodal way, we try to leverage this prior information

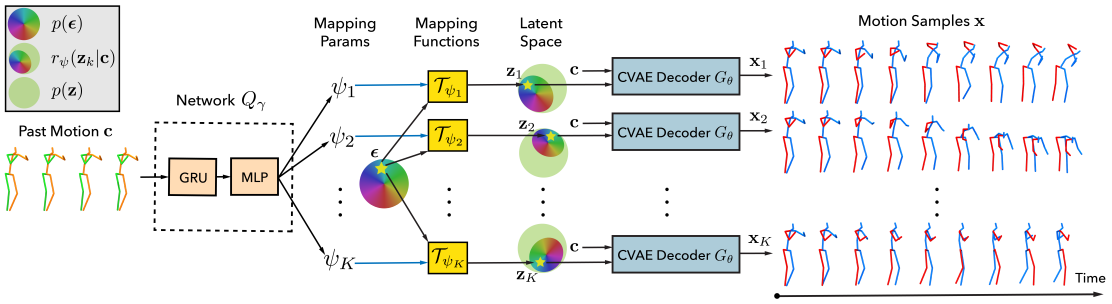


Figure 5.10: DLow architecture applied to human motion forecasting. The network Q outputs the parameters of the transformations that are applied to ϵ to obtain the k latent codes z_i. The multiple latent codes are fed to the generator to produce multiple and diverse outputs. From [42].

and also to change the parameters in order for the model to take more into account the latent vector and its influence on the prediction. Allowing the parameters to update makes it possible for more diverse predictions and predictions that are still plausible since the model has already been trained and is only being fine-tuned. In this phase, we only need the generator from Transformer GAN and the discriminator is discarded. We add the Q network of DLow after the encoder which takes in input the encoded past trajectory and outputs the parameters of the transformations to be applied to the new latent vector ϵ. In this way, the transformations are aware of the past trajectory history and are conditioned on it as the decoder is conditioned on it too. One of the main drawbacks of this sampling is that the numbers of samples to be generated is fixed and chosen at the start of the training. That is because the transformations of the vector ϵ are k and k needs to be fixed and known for training. Two models, one that generates $k = 20$ samples and the other $k = 5$ samples, will need two different trainings because the models will tune their predictions based on how many samples they generate.

The proposed method does not estimate the probabilities for each predicted trajectory. That is because the metrics used in literature, the TopK-ADE and TopK-FDE, do not consider any trajectory probability information and, for this reason, also the majority of the models present in literature do not estimate the probability of each of the K trajectories. Obviously, a model capable of such predictions would be preferable, but the literature benchmarking protocol and the fact that no ground truth probability is available for the trajectories makes this aspect also very difficult to be evaluated properly.

The overall architecture and flow of the sampling method is in Figure 5.11.

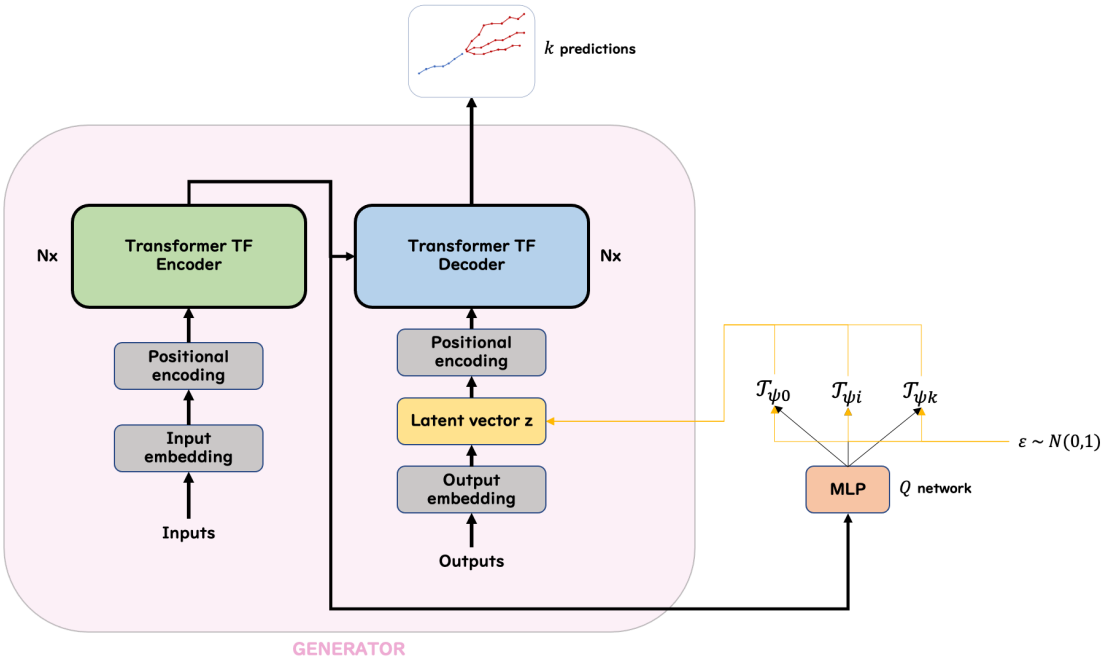


Figure 5.11: Proposed sampling method architecture, DLow with fine-tuning of the generator parameters. The Q network outputs the parameters that are used to transform the sampled vector ϵ in k latent vectors that generate k predictions.

To train the overall model, the generator and the new sampling section, a new loss is used which combines the DLow loss with the already used trajectory forecasting loss:

$$\mathcal{L} = \mathcal{L}_{trajectory} + \mathcal{L}_{diversity} + \mathcal{L}_{KL} \tag{5.15}$$

This loss is useful because it forces the generated samples to cover the ground truth trajectory, to differentiate the predictions and for the generated latent codes to follow a prior distribution. In particular, the three terms are defined as follows:

- $\mathcal{L}_{trajectory}$ is the pairwise distance loss of the standard model. The only difference is that in this case we have k predictions for each sample and we compute the distance loss only on the most close example to ground truth. Formally, the $\mathcal{L}_{trajectory}$ loss is defined as the minimum over k loss:

$$\mathcal{L}_{trajectory} = \min_{k} ||Y - \hat{Y}_k||_2 \tag{5.16}$$

  It is used to encourage sample diversity while covering also the ground truth trajectory. In fact, the model is incentivized to cover more possible outputs to cover the ground truth in all cases.

- The second part of the loss is used to explicitly account for diversity in the predictions:

$$\mathcal{L}_{diversity} = \exp\left(-\frac{1}{k^2}\sum_{i=1}^{k}\sum_{j=1}^{k}||\hat{Y}_j - \hat{Y}_i||_2\right) \tag{5.17}$$

  which essentially measures the distance between every pair of trajectories for the same input sample and averages it across the k predictions. The exponential is used to have a lower bound of zero for the loss. It measures the average distance between the k predictions.

- The last loss used is the KL divergence that measures the difference between the transformations applied on the original latent vector ϵ and the original prior $p(z)$ used in the Transformer GAN training. Since for $p(z)$ it was used a standard Normal $\mathcal{N}(0, I)$, this divergence loss imposes that the k generated latent vectors approximately follow the same distribution and forces the model to generate latent vectors similar to the ones already seen in the first training of Transformer GAN. It acts as a regularization term and, if the distribution p we want to approximate is $\mathcal{N}(0, I)$, as in our case, we can analytically compute it as:

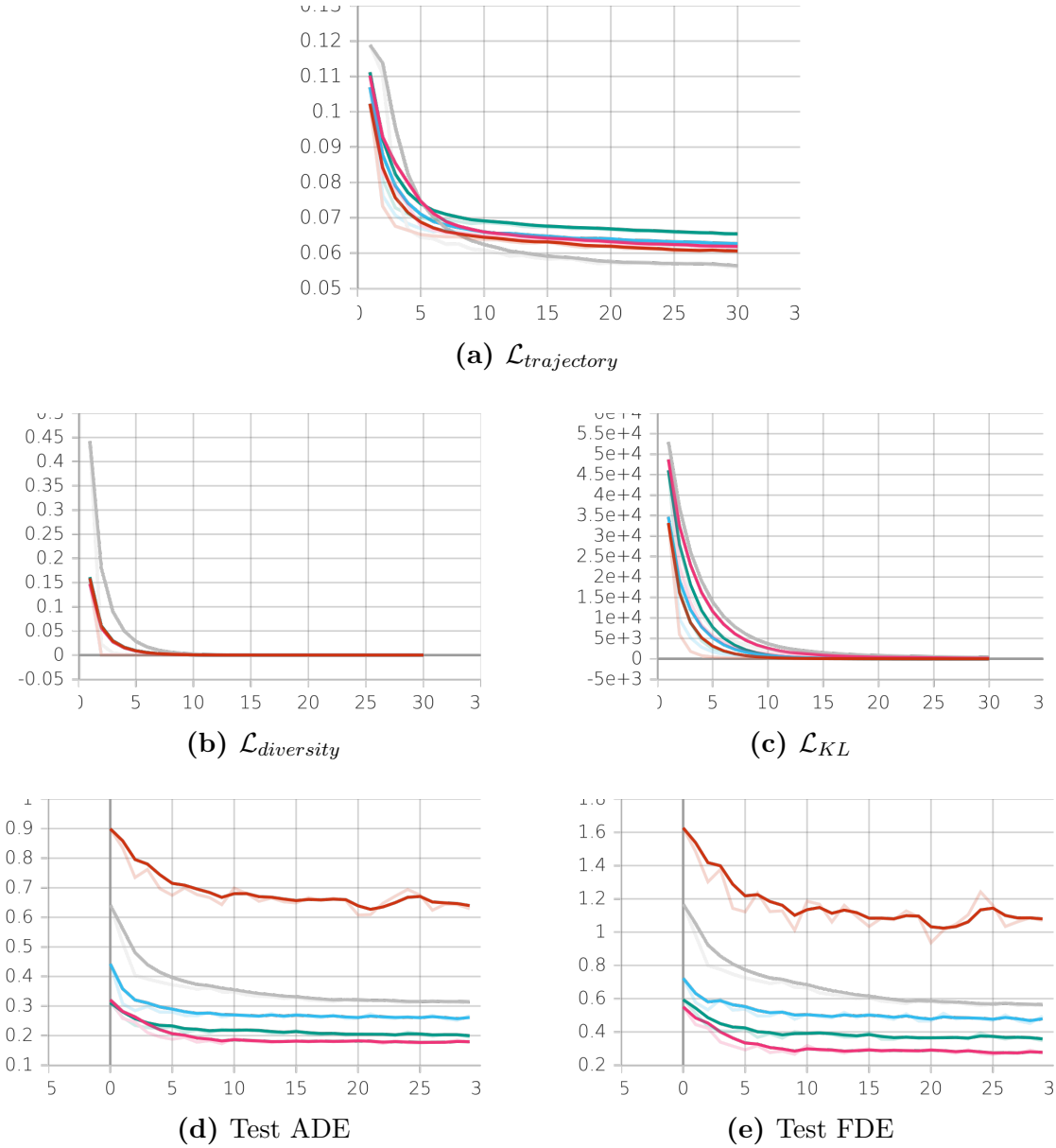
$$\mathcal{L}_{KL}(q||p) = -\frac{1}{2}[-\mu_q^T\mu_q - tr\{\Sigma_q\} + k + \log|\Sigma_q|] \tag{5.18}$$

  which tells us the divergence between the generated distribution q and the desired one. By minimizing this loss term we force q to follow our desired Normal distribution. It is important to notice that the parameters of q are outputted by the Q network in Figure 5.11.

Implementation details

All the models were trained for 30 epochs taking as input the previously trained Transformer GAN model instance. Adam optimizer was used with an initial learning rate of 0.0001 multiplied by 0.8 every 4 epochs. The learning rate was optimized in [0.001, 0.0005, 0.0001]. No other hyperparameter was optimized following a grid search because of time constraints. Batch size is set to 60. The Q network consists in an MLP which takes in input the encoded past trajectory and feed it to two consecutive layers of dimensions 512 and 256 respectively. The output of the Q MLP is then fed to two distinct linear layers with output dimension of $k \times z_{dim}$ that output the parameters of A and b used to transform the latent vector ϵ. The dimensions of the linear layers used in the Q network was set following the prior work of AgentFormer [8] which used DLow.
To reduce latency and speed up the training, the generator phase is divided in two parts, the first in which the past history is encoded (notice that the past history is the same for all k predictions) and the second in which this encoded sequence

is used to generate all the predictions. This approach is significantly faster than encoding k times the same input sequence as can be naively implemented.
The learning curves of the models are available in Figure 5.12.



(a) $\mathcal{L}_{trajectory}$

(b) $\mathcal{L}_{diversity}$

(c) $\mathcal{L}_{KL}$

(d) Test ADE

(e) Test FDE

Figure 5.12: Learning curves for the sampling method. ETH brown, Hotel magenta, UCY grey, Zara1 light blue and Zara2 green.

5.5.3 Results

The results of the stochastic models proposed are available in Table 5.7. There are both results prior and after the sampling method.

For Transformer GAN prior to the sampling method, it is evident that by increasing the number of drawn samples the results improve, but not by a large margin. In fact, the final results show that by drawing twenty samples we reach almost identical results of the original Transformer TF which is deterministic. This means that the samples that are drawn are all very similar to each other and do not bring much differentiation and improvements. The reason for this behaviour is one of the challenges of modelling multimodality which is that the model learns to cover the ground truth and most likely trajectory while the latent vector z is not taken very much in consideration while predicting. In fact, the pairwise distance loss used induces the generator output to follow the ground truth, learning the patterns that guides human motion but in a unimodal way. For this reason, the sampling method was proposed, which helps the model use its previous learned information while trying to differentiate the multiple predictions.

The effectiveness of the sampling method for the stochastic approach is evident. Both the 5 samples and the 20 samples versions outperform their prior to sampling counterparts. The 1 sample version is not reported because the model needs more than 1 sample for trajectory in order to be trained (because of the diversity loss). We can see how the results improve drastically from 5 to 20 samples, that is because the more the predicted trajectories, the more the possibility space is covered by those, indicating that the model differentiation between predictions is very good and effective.

Dataset	Transformer TF	Transformer GAN			Sampling method	
	Deterministic	1S	5S	20S	5S	20S
ETH	1.00/2.02	1.03/1.98	1.02/1.96	1.01/1.94	0.94/1.87	**0.61/0.94**
Hotel	0.33/0.65	0.35/0.74	0.34/0.72	0.34/0.71	0.29/0.54	**0.18/0.26**
UCY	0.55/1.19	0.57/1.21	0.55/1.15	0.53/1.11	0.59/1.24	**0.31/0.56**
Zara1	0.44/0.99	0.63/1.50	0.62/1.46	0.61/1.43	0.52/1.11	**0.25/0.45**
Zara2	0.34/0.75	0.39/0.83	0.36/0.72	0.33/0.66	0.42/0.78	**0.19/0.35**
avg	0.53/1.12	0.59/1.25	0.58/1.20	0.56/1.17	0.55/1.11	**0.31/0.51**

Table 5.7: Overall results of Transformer GAN prior and after the sampling method. The improved deterministic Transformer TF results are also shown for completeness. The number before S refers to the number of samples predicted. The metrics used for the stochastic models are the TopK-ADE and TopK-FDE with K equal to the number of samples drawn.

5.5.4 Predictions comparison before and after sampling method

In this section some examples are presented to show the capabilities and improvements of the model Transformer GAN before and after the sampling method was introduced. The examples shown here are difficult and critical examples in which some particular behaviour can be seen.
As we can observe from all Figures (5.13,5.14,5.15) in the top row, the Transformer GAN predictions are all unimodal and fail to capture the multimodality of the task. The k predictions are slightly different but they all follow the same direction. Differently, the predictions after the sampling refinement are very diverse and cover multiple possible modes in all the examples, and they cover multiple modes leveraging the past information and not just predicting random diverse trajectories.

In Figure 5.13 we see an example taken from Hotel dataset where the original Transformer GAN predicts a wrong trajectory while the model after the sampling method predicts very different trajectories that cover multiple modes (going straight, gently curve to the right or hard right curve). In this way the correct output mode is in the proposed ones.

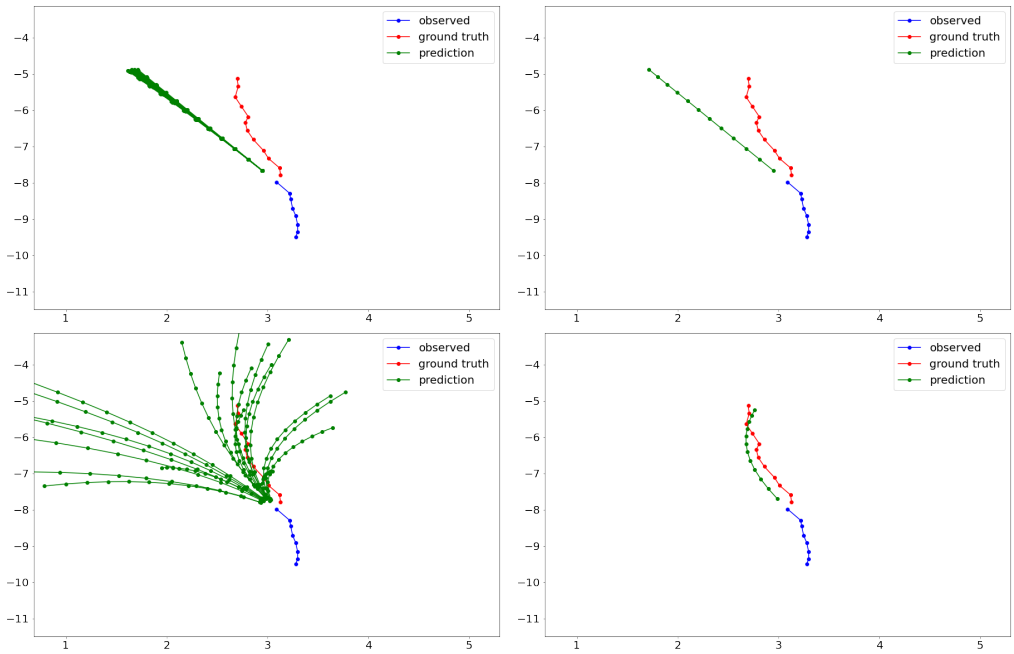


Figure 5.13: Comparison between TF GAN prior (top) and after (bottom) sampling method on an example from Hotel dataset. On the left 20 predicted trajectories, on the right the best one.

In Figure 5.14 another difficult example is presented from the UCY dataset. In this case the difficulty is that the observed trajectory is not much informative of the possible intent and future of the agent. In fact, the observed positions show an agent which is almost still. In this situation the possible outcomes are very uncertain and predicting the correct trajectory just from past history is almost a gamble. Original Transformer GAN does not capture the possible multiple outcomes and predicts intuitively the most likely which is to continue walking at the same speed, while the model after the refinement understands the uncertainty in the scene and tries to cover most of the possible outputs predicting some trajectories in all directions and some others that cover the possibility of the agent continuing to stand still.

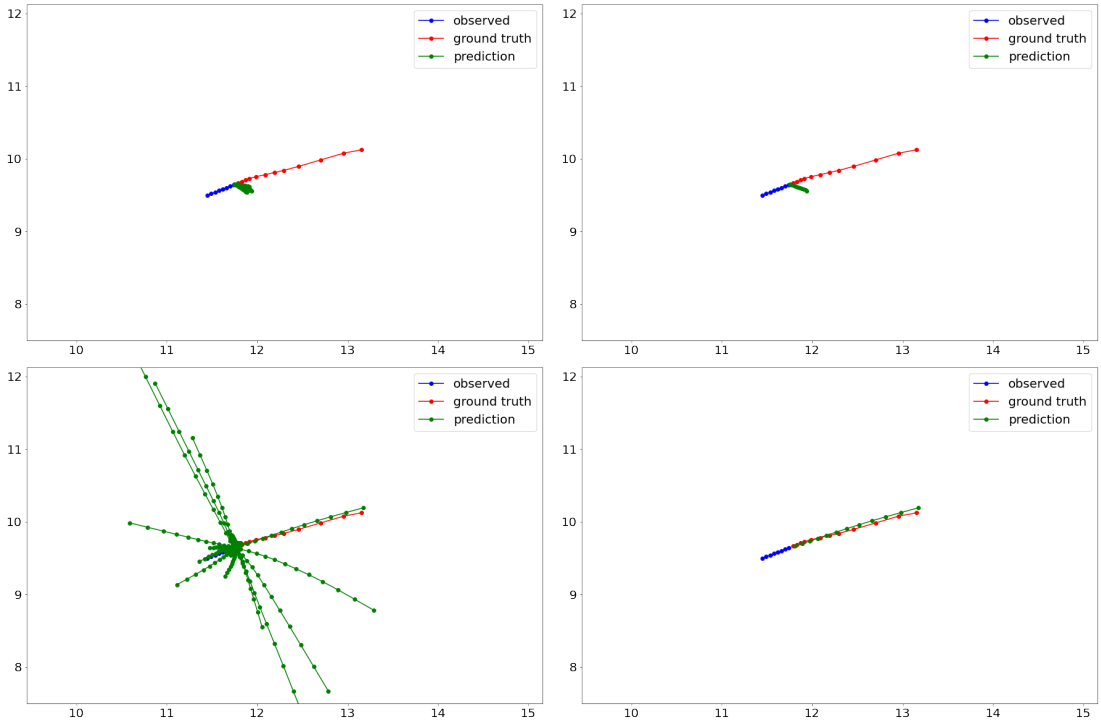


Figure 5.14: Comparison between TF GAN prior (top) and after (bottom) sampling method on an example from UCY dataset. On the left 20 predicted trajectories, on the right the best one.

Another particular aspect that can be seen from Figure 5.15 which is from the Zara1 dataset is the way the model after the sampling method is not only capable of covering multiple directions, but also different speeds of the agent. From the figure we can see that our final model predicts two set of trajectories if we consider the speed, one with lower velocities and the other with higher velocities. Even if the u-turn of the agent is not correctly predicted, the best prediction is much closer

to ground truth than the original Transformer GAN.

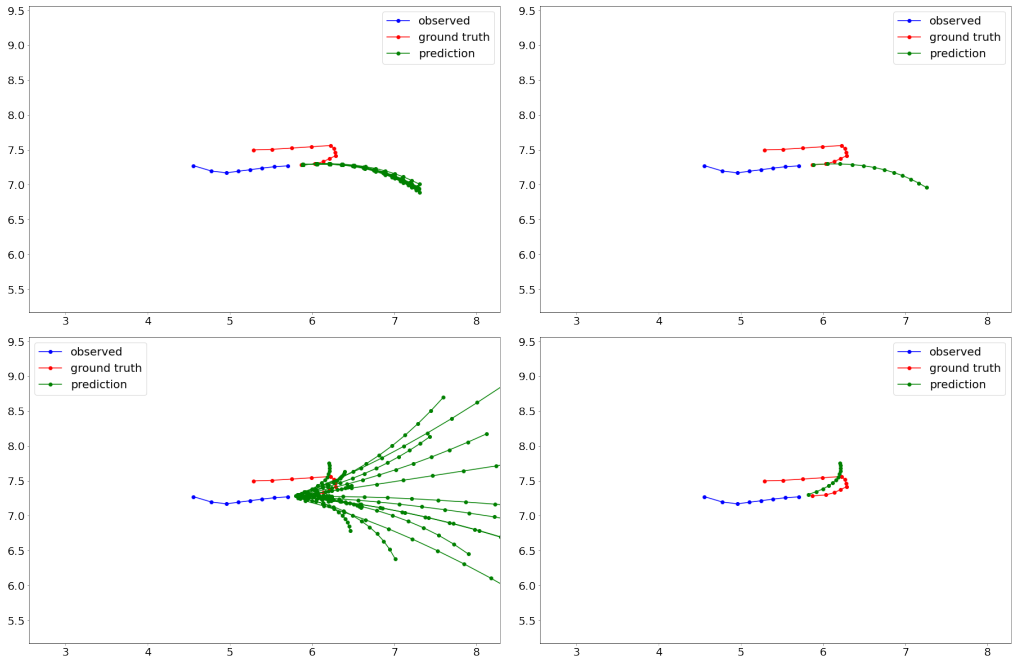


Figure 5.15: Comparison between TF GAN prior (top) and after (bottom) sampling method on an example from Zara1 dataset. On the left 20 predicted trajectories, on the right the best one.

5.5.5 Literature comparison

In Table 5.8 various results from literature are shown and compared to the Transformer GAN after the sampling method. We can observe how the model proposed performs slightly better than TF_q which is a version of Transformer TF quantized, in which the problem is defined as a classification task and the outputs are classes probabilities that can be sampled to get multiple results. The other model, SGAN-ind is the version of Social GAN that do not use social information. The proposed model after the sampling method outperforms SGAN-ind by a large margin indicating the goodness of a Transformer based model with respect to a LSTM model for this task. The other models results are reported as a reference and are not directly comparable to our proposed one since their inputs contain more information than just trajectory information, for example social or map information.

Model	ETH	Hotel	UCY	Zara1	Zara2	avg
Ours	**0.61/0.94**	**0.18/0.26**	**0.31/0.56**	0.25/0.45	0.19/0.35	**0.31/0.51**
TF_q [7]	**0.61**/1.12	**0.18**/0.30	0.35/0.65	**0.22/0.38**	**0.17/0.32**	**0.31**/0.55
SGAN-ind. [4]	0.81/1.52	0.72/1.61	0.60/1.26	0.34/0.69	0.42/0.84	0.58/1.18
SGAN [4]	0.87/1.62	0.67/1.37	0.76/1.52	0.35/0.68	0.42/0.84	0.61/1.21
AgentFormer [8]	0.45/0.75	0.14/0.22	0.25/0.45	0.18/0.30	0.14/0.24	0.23/0.39
Trajectron++ [5]	0.39/0.83	0.12/0.21	0.20/0.44	0.15/0.33	0.11/0.25	0.19/0.41
Social-BiGAT [48]	0.69/1.29	0.49/1.01	0.55/1.32	0.30/0.62	0.36/0.75	0.48/1.00
STAR [46]	0.36/0.65	0.17/0.36	0.31/0.62	0.26/0.55	0.22/0.46	0.26/0.53
SoPhie [49]	0.70/1.43	0.76/1.67	0.54/1.24	0.30/0.63	0.38/0.78	0.54/1.15
PECNet [10]	0.54/0.87	0.18/0.24	0.35/0.60	0.22/0.39	0.17/0.30	0.29/0.48

Table 5.8: Comparison between literature results of stochastic models using a TopK-ADE and TopK-FDE with K=20 samples. The results are taken from publications. Our model is the Transformer GAN generator after the sampling method. The first three models are the only comparable in terms of inputs as they consider only the trajectory information. The others use also map or social information

Chapter 6

Conclusions

In this thesis the pedestrian trajectory forecasting task is investigated by using deep learning methodologies. The task presents multiple challenges but for this work particular emphasis is placed in modelling the stochastic aspect of human motion. Furthermore, the problem is tackled using Transformer based architectures motivated by their good performances on sequence modelling tasks as trajectory forecasting is.

The initial model used in the experiments is Transformer TF [7], a Transformer based deterministic model in its original setting which does not use any social or map information. Despite its architecture simplicity, the model outperforms some other more complex solutions for the task. Experiments were carried out on the model where the input dimension was lowered making the model smaller with no significant impact on performance, then multiple losses and coordinates encodings were tried to observe the effect on training and lastly some data augmentation techniques were proposed which improved the original model performances while also having a smaller model thanks to a lower input dimensionality.
The second part of the experiments investigates the problem stochasticity. A new model proposal is made, Transformer GAN, based on Transformer TF but with the capability of making multiple predictions from a single observed past. The proposed generative model is a Generative Adversarial Network and can make multiple predictions thanks to a sampled latent vector. Then, a sampling method is introduced to diversify the predictions which drastically improves the stochastic performances of Transformer GAN.

The contributions of this thesis are a modified version of the original model Transformer TF with improved performances, thanks to a data augmentation technique, and smaller size, thanks to a lower dimensional input embedding. Then, a new model is proposed, Transformer GAN, to model the stochasticity of human behaviour thanks to a sampled latent code and a sampling technique is employed which drastically improves the model stochastic performances that reach

state-of-the-art results if considering models with same inputs as ours, meaning only trajectory history data. All the models and experiments were tested on the ETH-UCY datasets, a common benchmark used in literature to assess pedestrian trajectory forecasting models performances.

Future works could concentrate on adding more input information to make the predictions. This information could be social or map and scene data, to observe how adding such inputs can be beneficial to the model and in which occasions. In addition to this, also other metrics could be used to evaluate model performances such as metrics which quantify the collisions that occur in the predicted trajectories. These metrics could be useful to assess the real-world feasibility of the generated trajectories and would give an additional way to compare models, since distance metrics are close to saturation with current ones.

Bibliography

[1] Christoph Schöller, Vincent Aravantinos, Florian Lay, and Alois Knoll. *What the Constant Velocity Model Can Teach Us About Pedestrian Motion Prediction*. 2020. arXiv: 1903.07933 [cs.CV] (cit. on pp. 1, 7).

[2] Dirk Helbing and Péter Molnár. «Social force model for pedestrian dynamics». In: *Physical Review E* 51.5 (May 1995), pp. 4282–4286. ISSN: 1095-3787. DOI: 10.1103/physreve.51.4282. URL: http://dx.doi.org/10.1103/PhysRevE.51.4282 (cit. on pp. 2, 8).

[3] Alexandre Alahi, Kratarth Goel, Vignesh Ramanathan, Alexandre Robicquet, Li Fei-Fei, and Silvio Savarese. «Social LSTM: Human Trajectory Prediction in Crowded Spaces». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016 (cit. on pp. 2, 44–46).

[4] Agrim Gupta, Justin Johnson, Li Fei-Fei, Silvio Savarese, and Alexandre Alahi. *Social GAN: Socially Acceptable Trajectories with Generative Adversarial Networks*. 2018. arXiv: 1803.10892 [cs.CV] (cit. on pp. 2, 46, 66, 67, 81).

[5] Tim Salzmann, Boris Ivanovic, Punarjay Chakravarty, and Marco Pavone. *Trajectron++: Dynamically-Feasible Trajectory Forecasting With Heterogeneous Data*. 2021. arXiv: 2001.03093 [cs.RO] (cit. on pp. 2, 47, 64, 66, 81).

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL] (cit. on pp. 2, 28–31, 33, 48, 55).

[7] Francesco Giuliari, Irtiza Hasan, Marco Cristani, and Fabio Galasso. *Transformer Networks for Trajectory Forecasting*. 2020. arXiv: 2003.08111 [cs.CV] (cit. on pp. 2, 48, 49, 53, 55, 57, 81, 82).

[8] Ye Yuan, Xinshuo Weng, Yanglan Ou, and Kris Kitani. *AgentFormer: Agent-Aware Transformers for Socio-Temporal Multi-Agent Forecasting*. 2021. arXiv: 2103.14023 [cs.AI] (cit. on pp. 2, 49, 50, 64, 66, 75, 81).

[9] Parth Kothari, Sven Kreiss, and Alexandre Alahi. «Human Trajectory Forecasting in Crowds: A Deep Learning Perspective». In: *IEEE Transactions on Intelligent Transportation Systems* (2021), pp. 1–15. DOI: 10.1109/TITS.2021.3069362 (cit. on pp. 3, 5, 44).

[10] Karttikeya Mangalam, Harshayu Girase, Shreyas Agarwal, Kuan-Hui Lee, Ehsan Adeli, Jitendra Malik, and Adrien Gaidon. «It Is Not the Journey but the Destination: Endpoint Conditioned Trajectory Prediction». In: *arXiv preprint arXiv:2004.02025* (2020) (cit. on pp. 4, 81).

[11] Andrey Rudenko, Luigi Palmieri, Michael Herman, Kris M Kitani, Dariu M Gavrila, and Kai O Arras. «Human motion trajectory prediction: a survey». In: *The International Journal of Robotics Research* 39.8 (2020), pp. 895–935. DOI: 10.1177/0278364920917446 (cit. on pp. 6, 8).

[12] Kota Yamaguchi, Alexander C. Berg, Luis E. Ortiz, and Tamara L. Berg. «Who are you with and where are you going?» In: *CVPR 2011*. 2011, pp. 1345–1352. DOI: 10.1109/CVPR.2011.5995468 (cit. on p. 9).

[13] Alexandre Robicquet, Amir Sadeghian, Alexandre Alahi, and Silvio Savarese. «Learning Social Etiquette: Human Trajectory Understanding In Crowded Scenes». In: *Computer Vision – ECCV 2016*. Ed. by Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling. Cham: Springer International Publishing, 2016, pp. 549–565. ISBN: 978-3-319-46484-8 (cit. on p. 10).

[14] Frank Rosenblatt. «The perceptron: a probabilistic model for information storage and organization in the brain.» In: *Psychological review* 65 6 (1958), pp. 386–408 (cit. on pp. 11, 12).

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on pp. 11, 19).

[16] Alexander Amini. *MIT 6.S191 (2020): Introduction to Deep Learning*. 2020. URL: https://www.youtube.com/watch?v=njKP3FqW3Sk (cit. on p. 11).

[17] Antonio Rafael Sabino Parmezan. URL: https://www.researchgate.net/figure/Structure-of-Perceptron_fig2_330742498 (cit. on p. 12).

[18] Shruti Jadon. «Introduction to different activation functions for deep learning». In: *Medium, Augmenting Humanity* 16 (2018) (cit. on p. 13).

[19] IBM Cloud Education. *Neural Networks*. 2020. URL: https://www.ibm.com/cloud/learn/neural-networks (cit. on p. 14).

[20] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2017. arXiv: 1609.04747 [cs.LG] (cit. on p. 18).

[21] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG] (cit. on p. 20).

[22] Christopher Olah. *Understanding LSTM Networks*. 2017. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (cit. on pp. 22, 23).

[23] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. «Learning representations by back-propagating errors». In: *Nature* 323 (1986), pp. 533–536 (cit. on p. 21).

[24] P.J. Werbos. «Backpropagation through time: what it does and how to do it». In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560. DOI: 10.1109/5.58337 (cit. on p. 21).

[25] Y. Bengio, P. Simard, and P. Frasconi. «Learning long-term dependencies with gradient descent is difficult». In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166. DOI: 10.1109/72.279181 (cit. on p. 22).

[26] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. «On the Difficulty of Training Recurrent Neural Networks». In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. ICML'13. Atlanta, GA, USA: JMLR.org, 2013, III–1310–III–1318 (cit. on p. 22).

[27] Sepp Hochreiter and Jürgen Schmidhuber. «Long Short-term Memory». In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735 (cit. on p. 22).

[28] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL] (cit. on pp. 25–27).

[29] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. *Effective Approaches to Attention-based Neural Machine Translation*. 2015. arXiv: 1508.04025 [cs.CL] (cit. on p. 25).

[30] Jay Alammar. *The illustrated transformer*. 2018. URL: http://jalammar.github.io/illustrated-transformer/ (cit. on p. 32).

[31] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL] (cit. on p. 35).

[32] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. «GloVe: Global Vectors for Word Representation». In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: http://www.aclweb.org/anthology/D14-1162 (cit. on p. 35).

[33] Amirhossein Kazemnejad. *Transformer Architecture: The Positional Encoding*. 2019. URL: https://kazemnejad.com/blog/transformer_architecture_positional_encoding/ (cit. on pp. 35, 36).

[34] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML] (cit. on p. 37).

[35] Mehdi Mirza and Simon Osindero. *Conditional Generative Adversarial Nets*. 2014. arXiv: 1411.1784 [cs.LG] (cit. on pp. 37, 38).

[36] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2014. arXiv: 1312.6114 [stat.ML] (cit. on p. 38).

[37] Jeremy Jordan. *Variational autoencoders*. 2018. URL: https://www.jeremyjordan.me/variational-autoencoders/ (cit. on pp. 39, 41).

[38] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. «Learning Structured Output Representation using Deep Conditional Generative Models». In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/file/8d55a249e6baa5c06772297520da2051-Paper.pdf (cit. on p. 41).

[39] Matteo Lisotto, Pasquale Coscia, and Lamberto Ballan. *Social and Scene-Aware Trajectory Prediction in Crowded Spaces*. 2019. arXiv: 1909.08840 [cs.CV] (cit. on p. 45).

[40] Daksh Varshneya and G. Srinivasaraghavan. *Human Trajectory Prediction using Spatially aware Deep Attention Models*. 2017. arXiv: 1705.09436 [cs.LG] (cit. on p. 45).

[41] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL] (cit. on p. 48).

[42] Ye Yuan and Kris Kitani. *DLow: Diversifying Latent Flows for Diverse Human Motion Prediction*. 2020. arXiv: 2003.08386 [cs.CV] (cit. on pp. 50, 71, 73).

[43] Stefano Pellegrini, Andreas Ess, and Luc Van Gool. «You'll Never Walk Alone: Modeling Social Behavior for Multi-target Tracking». In: Sept. 2009, pp. 261–268. DOI: 10.1109/ICCV.2009.5459260 (cit. on p. 53).

[44] Alon Lerner, Yiorgos Chrysanthou, and Dani Lischinski. «Crowds by Example». In: *Comput. Graph. Forum* 26 (Sept. 2007), pp. 655–664. DOI: 10.1111/j.1467-8659.2007.01089.x (cit. on p. 53).

[45] Pu Zhang, Wanli Ouyang, Pengfei Zhang, Jianru Xue, and Nanning Zheng. «SR-LSTM: State Refinement for LSTM towards Pedestrian Trajectory Prediction». In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019 (cit. on p. 61).

[46] Cunjun Yu, Xiao Ma, Jiawei Ren, Haiyu Zhao, and Shuai Yi. *Spatio-Temporal Graph Transformer Networks for Pedestrian Trajectory Prediction*. 2020. arXiv: 2005.08514 [cs.CV] (cit. on pp. 64, 81).

[47] Soumith Chintala, Emily Denton, Martin Arjovsky, and Michael Mathieu. *How to train a GAN*. 2016. URL: https://github.com/soumith/ganhacks#authors (cit. on p. 71).

[48] Vineet Kosaraju, Amir Sadeghian, Roberto Martín-Martín, Ian Reid, S. Hamid Rezatofighi, and Silvio Savarese. *Social-BiGAT: Multimodal Trajectory Forecasting using Bicycle-GAN and Graph Attention Networks*. 2019. arXiv: 1907.03395 [cs.CV] (cit. on p. 81).

[49] Amir Sadeghian, Vineet Kosaraju, Ali Sadeghian, Noriaki Hirose, S. Hamid Rezatofighi, and Silvio Savarese. *SoPhie: An Attentive GAN for Predicting Paths Compliant to Social and Physical Constraints*. 2018. arXiv: 1806.01482 [cs.CV] (cit. on p. 81).