UNIVERSITY OF BUCHAREST

FACULTY OF
MATHEMATICS AND
COMPUTER SCIENCE

COMPUTER SCIENCE SPECIALIZATION

Bachelor's thesis

SIMULATION ALGORITHMS FOR HUMAN AND CAR BEHAVIOR

Author
Andrei-Eusebiu Blahovici

Scientific coordinator
Assoc. Professor. Dr. Ciprian Ionut Paduraru

Bucharest, June 2022

Abstract

Self-driving cars have got a lot of attention in the past years, as the technology around them has evolved tremendously. Even though current self-driving cars have fairly good autonomy, there are still challenges that have to be overcome in order to have fully autonomous cars in the future.

This thesis aims to address the problem of motion prediction for autonomous agents. At first, I am going to review existing state-of-the-art deep learning approaches for this problem, then I will show how these solutions might be tested qualitatively in a simulation application such as CARLA.

Contents

Chapter 1

Introduction

1.1 Background

The technology in the industry of autonomous vehicles is already being under development by multiple companies, for all types of cars. Fully autonomous systems are still being tested to this day, even though partially autonomous systems have been available for a few years now.

The mechanisms that are involved in making self-driving cars have been the target of a lot of research and development efforts from a variety of car producers, universities, and research centers since the 1980s.

It is expected that autonomous vehicles are going to drastically redefine the future of transportation. Though, there are still a lot of technical difficulties that need to be solved before the world might fully enjoy the benefits of using fully autonomous cars.

The hardware used in the making of self-driving vehicles is said to be ready for production from a functionality point of view, whereas the software is not so much, as we still do not have really good solutions for the more difficult problems that come with such fully autonomous systems.

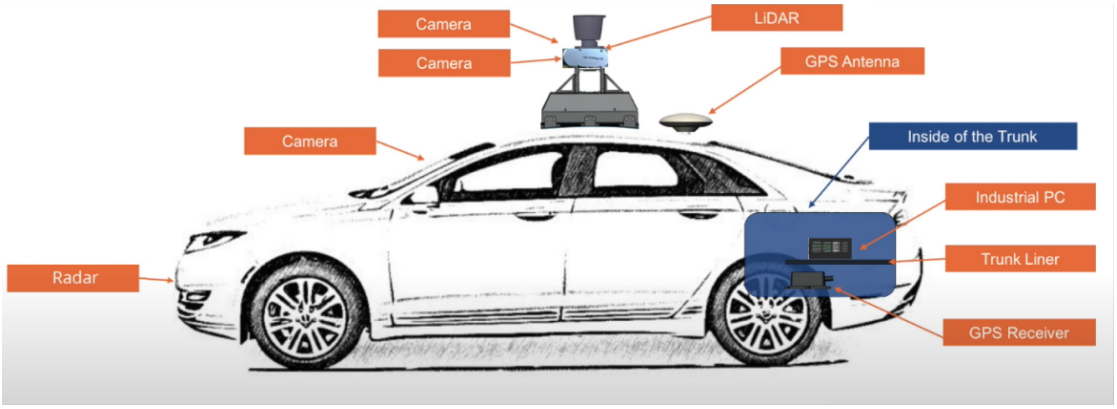

Figure 1.1: This figure resembles the hardware components that are used in making self-driving cars. This picture has first been presented in the paper Motion Prediction on Self-driving Cars: A Review [14].

1.2 Overview of self-driving cars

Creating self-driving cars, especially fully autonomous ones, is exceptionally difficult not only from a software standpoint but also from a hardware one.

At first, the car needs to identify the surroundings and its position relative to those. This step is called perception and is achieved through multiple types of sensors and cameras that need to assist the car. The most common hardware used in creating autonomous vehicles and how they are usually positioned on the car can be seen in Fig.1.1. The vehicle distinguishes relevant objects from the environment by creating highly detailed semantic maps using the images taken by the cameras, while the Radar(Radio Detection and Ranging) and the LiDAR(Light Detection and Ranging) are used to measure the distance to the surrounding objects.

Secondly, using the data collected in the previous step, the car has to predict the future trajectory of the surrounding cars, pedestrians and cyclists. As for today, this is still a major problem and represents a bottleneck in the development of fully autonomous vehicles.

At last, having the environment data and possible trajectories for each agent nearby, the vehicle should be able to plan its next steps according to this information.

1.3 Thesis objective

This thesis focuses on the problem of predicting the motion of surrounding agents such as other vehicles, pedestrians, and cyclists, using data from the perception module. It is a really difficult problem to overcome, because of its uncertain nature. In specific cases, not even humans can predict what is coming next on the road, but the systems used in fully autonomous vehicles must be right all the time, or at least have a safety mechanism in case some things go wrong.

Preliminary studies tried to use reinforcement learning to address this problem. The issue this approach faces is that training the model on a single scenario is slow and the training needs to be done on a lot of scenarios. To fix this, recent studies tried to use pre-trained deep learning models trained on large datasets, approach that shows state-of-the-art results so far.

In this thesis, I am going to focus on the deep learning approach. Also, since existing work for this solution focuses more on quantitative results, I am also going to show a way these models can be tested qualitatively in the CARLA Simulator, which is a simulator designed specifically for autonomous driving research.

The whole code I wrote for this project can be found at this link:
https://github.com/Dawlau/agents-motion-prediction

1.4 Personal Motivation

Self-driving cars are a really important topic as of today and they have the potential to shape the future of transportation globally. Considering this, I wanted to further my expertise in this technologically intense domain and get a feeling of what is the current status in the development of these types of vehicles.

The research I have done in this direction taught me a lot about how self-driving cars currently work, complex deep learning algorithms, how to handle really large datasets, and how to work with a simulator specialized in the research of autonomous driving.

Chapter 2

Related work

2.1 Waymo Open Dataset for Motion prediction

2.1.1 Description

The Waymo Open Dataset for Motion prediction is a really large dataset and it contains real-life scenarios that have been gathered by the Waymo Self-driving Car. Each scenario contains 9 seconds of data sampled at 10Hz. For every sample, there are coordinates, types, states, etc. describing each object in the scene such as agents, the road, and traffic lights.

2.1.2 Proposed metrics

The aim of the competition is to get the best overall results according to some metrics. Every single one of the following metrics is computed at the timestamps 3, 5, and 8 respectively. I am going to briefly describe those metrics in the rest of this section. These metrics are going to be relevant throughout this whole thesis.

Notations used in defining the metrics:

1. K - the number of predicted trajectories
2. T - the time horizon
3. G - a set of N agents

4. $\{(l_G^1, s_G^1), (l_G^2, s_G^2), \ldots, (l_G^K, s_G^K)\}$ - the set of predicted trajectories for all agents, where l_G^i is the likelihood of the trajectory i whereas s_G^i is the actual trajectory predicted

5. $\hat{s}_G^t$ - ground truth trajectory at time step t.

Minimum Average Displacement Error

This metric computes the L2 norm between the ground truth trajectories and the predicted ones, averaged across the time horizon.

$$minADE(G) = \min_i \frac{1}{T} \sum_{t=1}^{T} ||\hat{s}_G^t - s_G^{it}||_2$$

Minimum Final Displacement Error

MinFDE is similar to computing minADE at a fixed time step T

$$minFDE(G) = \min_i ||\hat{s}_G^t - s_G^{it}||_2$$

Miss Rate

A ground truth trajectory is considered to be missed when none of the predicted trajectories fall within a longitudinal and latitudinal threshold of the ground truth trajectory.

The Miss Rate is the total number of misses divided by the number of objects that predictions have been made for.

Overlap rate

An overlap is defined as the number of times the trajectory with the maximum likelihood has overlapped with objects from the scene over multiple time steps.

The overlap rate is the total number of overlaps over the total number of objects.

mAP

To compute the mAP metric, the trajectories predicted for an object are first sorted in descending order by their likelihood. Every trajectory classified as a miss is assigned a false positive value, and a true positive otherwise. Only the trajectory with the highest likelihood can be considered a true positive.

Using these labels, the precision and recall are being computed, and finally, the mAP metric for an object is calculated using the area under the precision-recall curve as described by Everingham, Mark, Van Gool et al. [6].

2.2 ReCoAt

2.2.1 Problem formulation

As presented by Zhiyu Huang, Xiaoyu Mo, and Chen Lv [20], the general mathematical formulation of the problem would be to try to predict a number of future trajectories for a specific agent, using some information about the surroundings. Formally, we need to find a function that takes an input X, represented by the historical states over a time period T_h of the surrounding agents and the environment context, and output the set $\hat{Y}$ of the K possible trajectories for the target agent. We can describe the input and the output using the following notations:

$$X = \{S_0, S_1, \ldots, S_N, \mathcal{M}\}$$
$$\hat{Y} = \{(x_j^t, y_j^t) \mid t \in \{t_0 + 1, \ldots, t_0 + T_f\}\}_{j=1}^{K}$$

Explanation of the notations used:

$S_i = \{s_i^{t_0 - T_h + 1}, s_i^{t_0 - T_h + 2}, \ldots, s_i^{t_0}\}$, the states of agent i

s_i^t is the state of the agent i at time step t

$\mathcal{M}$ is the environment information such as lanes, crosswalks etc.

t_0 is the current time step

2.2.2 Dataset pre-processing

In this paper, the dataset used is the Waymo Open Dataset for Motion Prediction dataset.

The state s_i^t is encoded as (x, y, v_x, v_y, θ), where (x, y) are the coordinates of the agent, (v_x, v_y) are the velocities for the O_x and O_y axis respectively, and θ is the heading angle of the agent. Also, only the closest 10 agents that are in the proximity of 30 meters of the target agent are taken into consideration. If there are not enough such agents, the tensor that keeps the encodings of the states is padded.

The environment context is represented as an image in the format bird's eye view, which is a photo of the scene as if there were a camera above. A few examples of rasters created for environment encoding can be seen in Fig. 2.1. In order to eliminate certain degrees of freedom in the environment representation, the image is always rotated and shifted such that the target agent is positioned at $(1/5, 1/2)$ of the image.

2.2.3 ReCoAt model architecture

The entire architecture of the model can be seen in Fig.2.2. The environment context is passed through a pre-trained ResNet50 [9] encoder and a linear layer. The states of the target agents are encoded using a trajectory encoder made with a 1D convolutional layer and a LSTM layer [10].

The states are passed through a trajectory encoder with a distance attention module. This module takes into consideration, for each agent, how important it is based on its distance to the target agent. Agents closer to the target agent should have a higher score than the ones that are farther. The formula for the attention layer is given by:

$$f_{att}(key_i, query) = \frac{\alpha}{\sqrt{(x_i - x_0)^2 + (y_i - y_0)^2}},$$

Where α is a hyperparameter, key_i is the position of agent i and $query$ represents the position of the target agent.

Also, there is a score that is calculated for each agent based on the outputs of

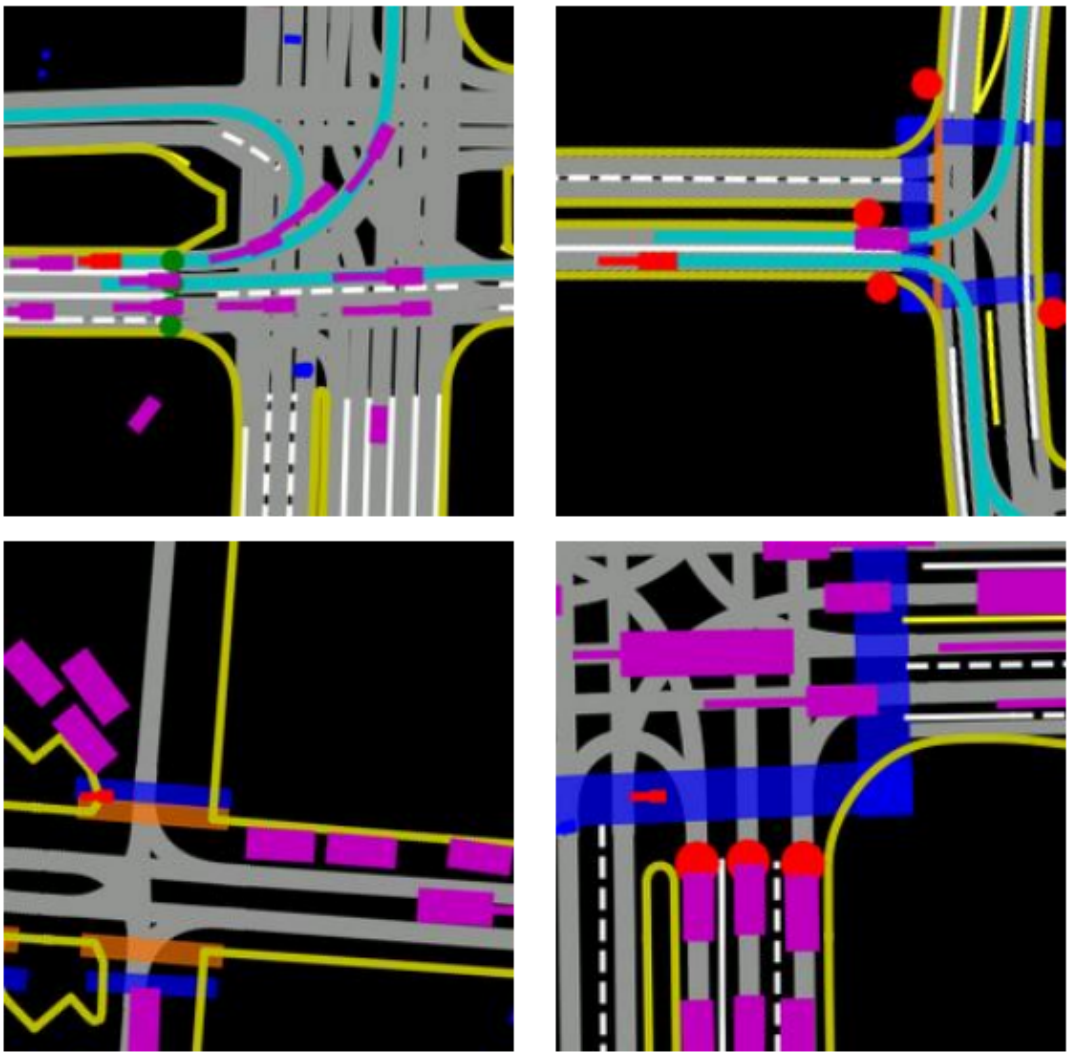

Figure 2.1: This photo shows 4 environment representations for different scenarios, used in the ReCoAt Motion Prediction Framework. The red rectangle represents the target agent and the purple ones the surrounding agents. The tails of each agent represent the historical positions. It is also worth mentioning that the red circles that are on the side of the roads are the stop signs, whereas the red or green circles that are actually on the road, represent the traffic lights

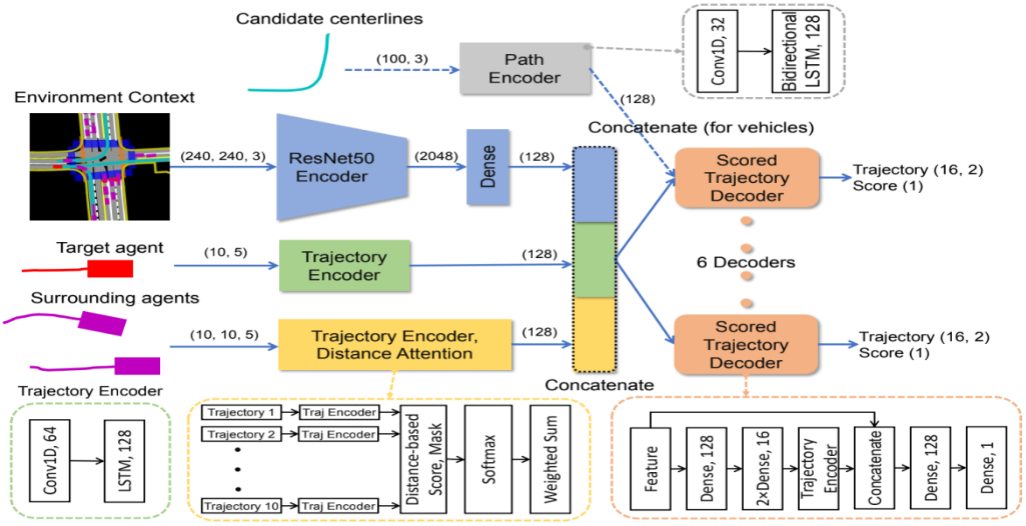

Figure 2.2: Here, you can see the overall architecture proposed in ReCoAt: A Deep Learning Framework with Attention Mechanism for Multi-Modal Motion Prediction [20], how the attention mechanisms and trajectory encoders are implemented, and what layers they use

the function above, using the softmax function:

$$\beta_i = \frac{e^{f_{att}(key_i, query)}}{\sum_j e^{f_{att}(key_j, query)}}$$

Finally, the attention module outputs a weighted sum given by the following formula:

$$out = \sum_i \beta_i \cdot value_i$$

2.2.4 Multi-task loss function

The part of the loss function that assesses the importance of the trajectory is really similar to minADE, the only difference being that each term in the summation is multiplied with a weight that adapts through time steps. The mathematical formula for this is the following:

$$\mathcal{L}_{traj} = \min_{j \in \{1,\dots,K\}} \frac{1}{T_f} \sum_t w_t^v ||(x_{gt}^t, y_{gt}^t) - (x_j^t, y_j^t)||$$

Where (x_{gt}^t, y_{gt}^t) are the ground truth coordinates at time step t and the weights adapt through time following the formula:

$$w_t^v = w_t \cdot w^v, w_t = 0.5t, w^v = max(1, 4 - 0.2v)$$

Besides the term involving the trajectories predicted, the loss function also takes into consideration the likelihoods of them, using the cross-entropy loss to measure the difference between ground truth probability and the estimated one:

$$\mathcal{L}_{score} = \mathcal{L}_{CE}(p_{gt}, p)$$

Where the ground truth probability is calculated by taking the L2 norm between the last positions of the ground truth and predicted trajectories and applying the softmax function. Mathematically, we can easily describe this process as:

$$p_{gt} = \frac{e^{-||s^{T_f} - s_{gt}^{T_f}||_2}}{\sum_j e^{-||s_j^{T_f} - s_{gt}^{T_f}||_2}}$$

The final loss function proposed in this paper is:

$$\mathcal{L} = \mathcal{L}_{score} + \lambda \mathcal{L}_{traj}$$

Where λ is a hyperparameter.

Method	minADE (m) ↓	minFDE (m) ↓	Miss Rate ↓	Overlap Rate ↓	mAP ↑
LSTM	1.0065	2.3553	0.3750	0.1898	0.1756
Ours	**0.7703**	**1.6668**	**0.2437**	**0.1642**	**0.2711**

Figure 2.3: This figure shows the quantitative results obtained by the ReCoAt model, in the Waymo Open Dataset for Motion Prediction 2021 competition, compared to the LSTM baseline proposed by the Waymo team

2.3 MotionCNN

2.3.1 Data rasterization

In contrast to the ReCoAt framework presented in the previous section, Konev Stepan, Brodt Kirill, and Sanakoyeu Artsiom [11] decided to employ a simpler deep learning model that is only based on a CNN backbone and a fully connected layer to generate trajectories and their confidences.

The input is a 224x224x25 image where the first 3 channels are the rgb representation of the environment like in ReCoAt, using the bird's eye view representation, the next 11 channels comprise the historical data of the target agent, and finally, the last 11 channels encode the historical data of the surrounding agents. An example of a raster is shown in Fig. 2.4.

A major bottleneck in training this model is the amount of time needed to process the rasters for all scenarios and all types of agents. For this, the authors decided to pre-process the rasters as compressed npz files prior to training, using multi-threading.

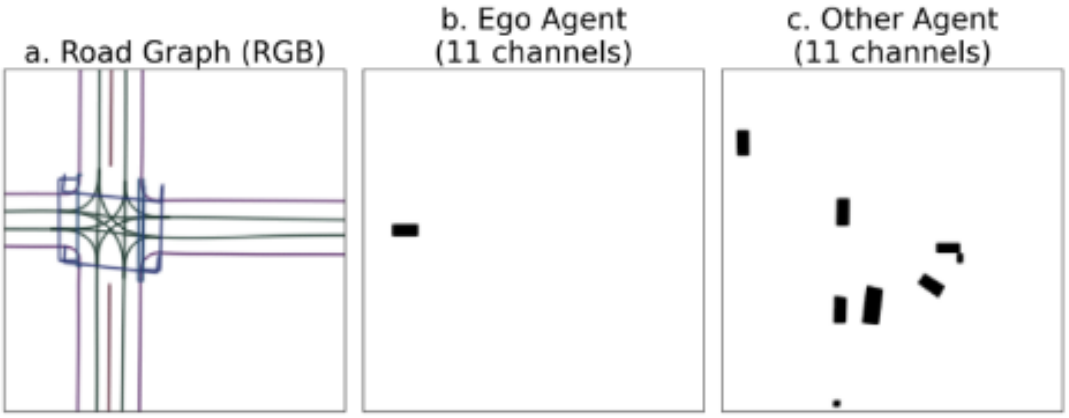

Figure 2.4: An example of a 25 channel raster used as input for the MotionCNN model

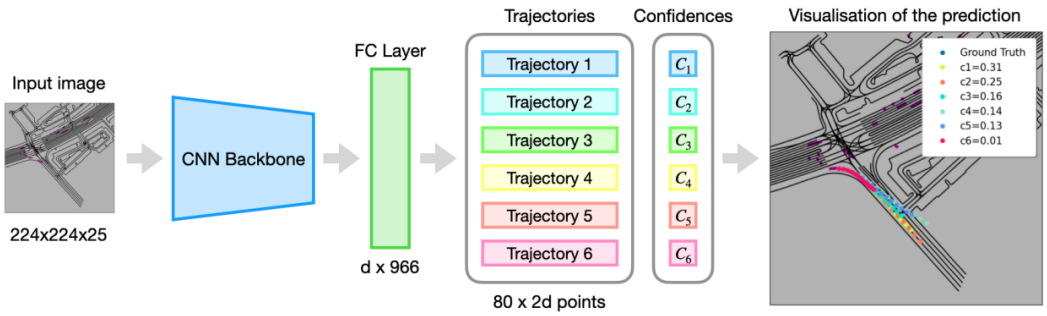

Figure 2.5: This figure represents the MotionCNN architecture. The CNN backbone can be either ResNet-18 or Xception71

2.3.2 MotionCNN architecture

As already mentioned, this architecture is much simpler than the one employed in the ReCoAt framework, but it still presents state-of-the-art results. The CNN backbones used in these experiments are ResNet-18 [9] and Xception71 [3]. An overview of the MotionCNN architecture can be seen in Fig.2.5.

2.3.3 Loss function for probabilistic modeling

A notable difference from the previously presented loss function is that instead of using a loss function more frequent with regression tasks, this paper employs a probabilistic modeling of the loss. The trajectories are modeled as a mixture of normal distributions. The network outputs the mean μ of the distribution while the covariance is kept as the identity matrix. With this architecture in place, the loss function used is the negative log-likelihood across all predicted trajectories. Formally, given the ground truth trajectory and the predicted trajectories, we can mathematically write the loss function as follows:

$$L = -logP(X^{gt}) = -log \sum_k c_k \mathcal{N}(X^{gt}; \mu = X_k, \Sigma = I)$$
$$= -log \sum_k c_k \prod_{t=1}^{T} \mathcal{N}(x_t^{gt}; x_{k,t}, 1)\mathcal{N}(y_t^{gt}; y_{k,t}, 1)$$
$$= -log \sum_k e^{log(c_k) - \frac{1}{2}\sum_{t=1}^{T}(x_t^{gt} - x_{k,t})^2 + (y_t^{gt} - y_{k,t})^2}$$

Notations used:

$X^{gt} = [(x_1, y_1), \ldots, (x_T, y_T)]$ is the ground truth trajectory

$X_k = [(x_{k,1}, y_{k,1}), \ldots, (x_{k,T}, y_{k,T})], k = 1 \ldots K$ are the K predicted trajectories

$\mathcal{N}(\cdot; \mu, \Sigma)$ is the normal distribution with mean μ and covariance Σ

	Method	mAP	Min ADE	Min FDE	Miss Rate	Overlap Rate
Test	Waymo LSTM baseline [1]	0.1756	1.0065	2.3553	0.3750	0.1898
	ReCoAt (2^{nd} place) [12]	0.2711	0.7703	1.6668	0.2437	0.1642
	DenseTNT (1^{st} place) [9]	**0.3281**	1.0387	1.5514	**0.1573**	0.1779
	MotionCNN-Xception71 (Ours)	0.2136	**0.7400**	**1.4936**	0.2091	**0.1560**
Val	MotionCNN-ResNet18 (Ours)	0.1920	0.8154	1.6396	0.2552	0.1605
	MotionCNN-Xception71 (Ours)	0.2123	0.7383	1.4957	0.2072	0.1576

Figure 2.6: Results obtained by the MotionCNN model in comparison to the models that ranked in the first two places at the Waymo Open Dataset competition

2.4 DenseTNT

2.4.1 Sparse context encoding

In contrast to other methods used for approaching this problem that rasterize the input and extract relevant features using CNNs, Gu Junru, Sun Chen, and Zhao Hang [8] propose a way of modeling the data sparsely. This is achieved by taking the global coordinates of all objects in the scene and encoding them as polylines.

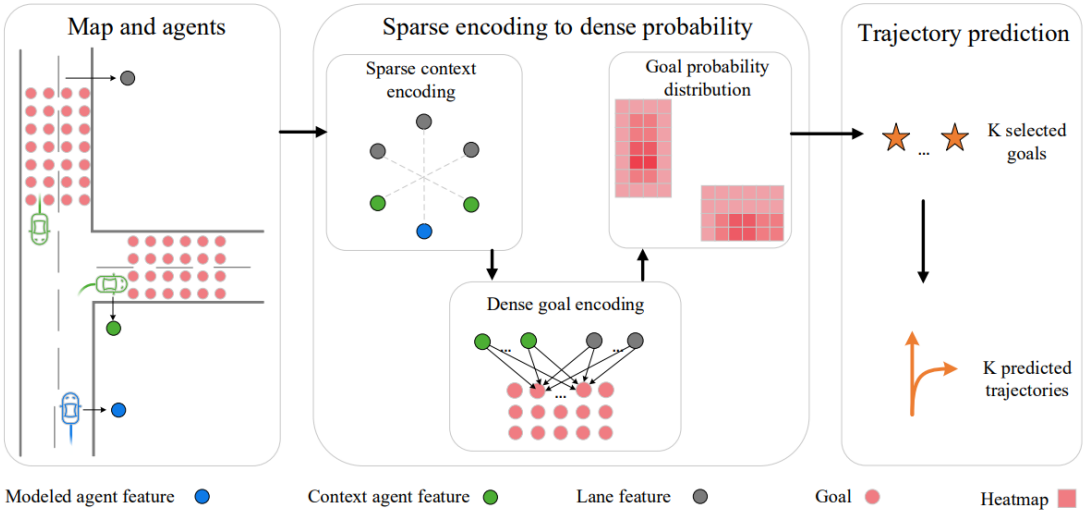

Figure 2.7: This figure presents the workflow in the DenseTNT model i.e. from data sparse encoding to predicting full trajectories

2.4.2 Estimating the probability of dense goals

After encoding the input in a sparse manner, the model takes each point on the road, sampled at a fixed sampling rate, and then calculates for each point, the likelihood that this can be a goal. In order to extract relevant features of each point, an attention mechanism based on the lane coordinates is used.

Let F_i be the features of the i-th goal, obtained from a 2-layer MLP which's input is the 2D coordinates of the goal. The attention mechanism is defined mathematically as follows:

$$Q_i = F_i W^Q, K = LW^K, V = LW^V$$

$$A_i(Q_i, K, V) = softmax(\frac{Q_i K^T}{\sqrt{d_k}})V$$

Notations used in describing the attention mechanism:

$W_Q, W_K, W_V \in \mathbb{R}^{d_h X d_k}$ – matrices for a linear transformation which can be implemented as a single layer MLP with no bias and activation function

d_k – dimension of the k-th attention head

F – feature matrix of the i-th goal

L – feature matrix of the agents and lanes

Finally, the score of the i-th goal is calculated using the softmax across all attention scores:

$$\phi_i = \frac{e^{g(A_i)}}{\sum_{n=1}^{N} e^{g(A_n)}}$$

Where function $g(\cdot)$ is yet another 2-layer MLP.

Goal selection is done via the non-maximum suppression algorithm, which takes the goals with the highest likelihood iteratively and removes the goals that are really close to it. This process is repeated until the number of trajectories is met.

Furthermore, for each goal selected, the trajectory is completed via a method fully described in Ronald J. Williams and David Zipser [19].

2.4.3 Multi-head loss

In this paper, a multi-head loss is used at training time, one head is used for training the probability estimation part, using the cross-entropy loss, while the other head is used to calculate how good the entire trajectory built is, using the average of the l1 norm. The 2 loss heads are described below.

$$\mathcal{L}_{goal} = \sum_i \mathcal{L}_{CE}(\phi_i, \psi_i)$$

Where $\mathcal{L}_{CE}$ is the cross-entropy loss, ϕ_i is the probability of the i-th goal and ψ_i is the ground truth probability of the goal, which is 1 for the closest goal to the real one and 0 for the others.

$$\mathcal{L}_{completion} = \sum_{t=1}^{T} \mathcal{L}_{reg}(\hat{s}_t, s_t)$$

Where $\mathcal{L}_{reg}$ is the l1 norm, $\hat{s}_t$ and s_t are the 2D coordinates of the trajectory at time step t of the ground truth and predicted ones respectively.

Method	minADE	minFDE	Miss Rate	Overlap Rate	mAP
DenseTNT 1^{st} (Ours)	1.0387	1.5514	0.1573	0.1779	**0.3281**
TVN 2^{nd}	0.7558	1.5859	0.2032	**0.1467**	0.3168
Star Platinum 3^{rd}	0.8102	1.7605	0.2341	0.1774	0.2806
SceneTransformer	**0.6117**	**1.2116**	**0.1564**	0.1473	0.2788
ReCoAt	0.7703	1.6668	0.2437	0.1642	0.2711
AIR	0.8682	1.6691	0.2333	0.1583	0.2596
SimpleCNNOnRaster	0.7400	1.4936	0.2091	0.1560	0.2136
CNN-MultiRegressor	0.8257	1.7101	0.2735	0.1640	0.1944
GOAT	0.7948	1.6838	0.2431	0.1726	0.1930
Waymo LSTM baseline	1.0065	2.3553	0.3750	0.1898	0.1756

Figure 2.8: DenseTNT obtained the first place in the Waymo Open Dataset for Motion prediction competition and this picture shows the metrics obtained by it compared to other models that competed

2.5 TPNet

2.5.1 TPNet architecture

Similar to MotionCNN, the TPNet architecture uses as input to their model a raster in the format of bird's eye view that encodes the historical states of the agents in the scene along with the environment information. Although, what is new in the paper proposed by Liangji Fang, Qinhong Jiang, Jianping Shi and Bolei Zhou [7] is that instead of regressing the trajectories with a fully-connected layer, they only regress the final endpoint of a trajectory, and sample the rest of the points according to the equation of a cubic curve. An overview of the proposed architecture is shown in Fig. 2.9.

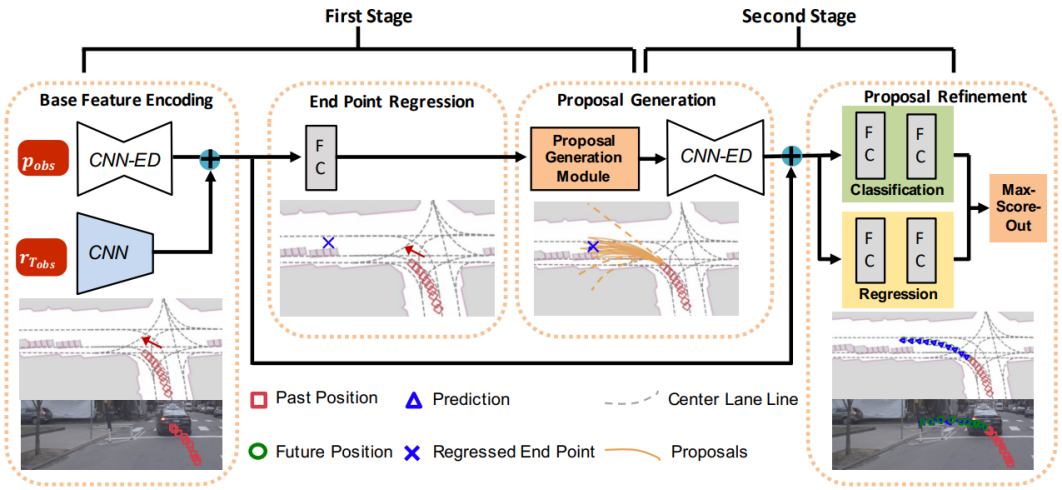

Figure 2.9: Overview of the neural network architecture proposed for TPNet. Each module also contains a 2D visualization of the results after every step

2.5.2 Generating proposals

The first way of generating proposals presented in this paper takes into consideration only the historical states of agents. To do so, the model uses as reference 3 parameters i.e. the historical positions, the regressed end-point, and a distance γ, which represents the distance between the curve of the trajectory and the midpoint of the segment between the last historical state and endpoint. A visual representation of the idea is presented in Fig. 2.10.

In order to obtain multiple trajectories following this idea, the model generates a set of N^2 of 2D points around the regressed endpoint, by iterating through 2 pointers i and j:

$$p_{ep} = \{(x_e + interval * i, y_e + interval * j)\}_{i,j \in [-N/2, N/2]}$$

Having a set of probable endpoints, a trajectory is generated for each, by varying the parameter γ:

$$proposals = \{f(p_{obs}, p'_{ep}, \gamma)\}$$

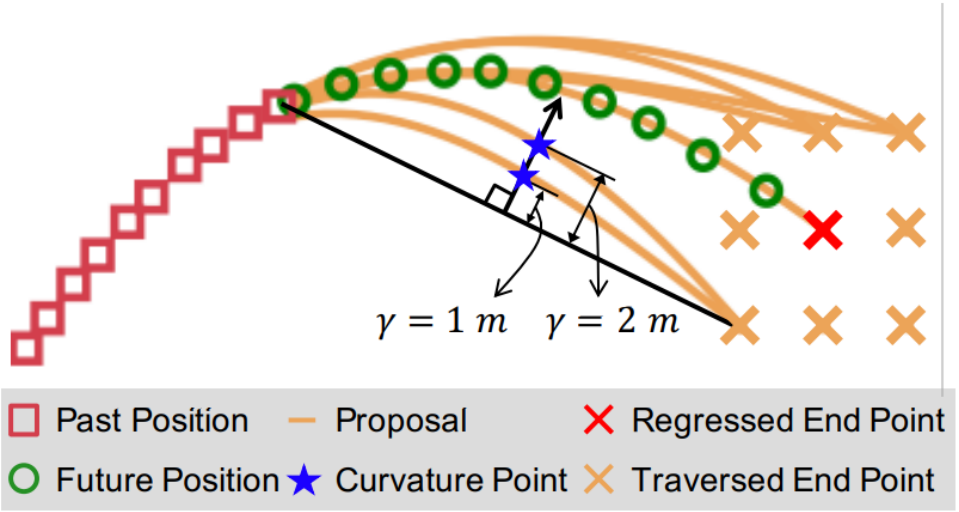

Figure 2.10: A visualization of the algorithm used for proposal generation using the crooked angle γ

Notations used in proposal generation:

p_{ep} – the set of possible endpoints

$p'_{ep} \in p_{ep}$ – one of the possible endpoints

$interval$ – a real number that represents the distance between two consecutive possible endpoints

p_e – the regressed endpoint

(x_e, y_e) – the coordinates of p_e

N – the size of the grid the possible endpoints are generated in

$f(\cdot)$ – cubic polynomial fitting function

In the case that road information is available, the predicted trajectories are not generated based on a regressed endpoint, as this might lower the diversity. Rather, based on basic information about the road, a displacement along the reference line is generated, which then is used to create a possible endpoint for a trajectory.

But, for actual trajectory generation, the process is the same as the one described above.

2.5.3 Classification and refinement of proposed trajectories

The TPNet architecture assigns first a label of "good" or "bad" to each proposed trajectory based on the average distance from the ground truth trajectory. The formula for the average distance is:

$$AD = \frac{1}{N} \sum_{i=1}^{N} ||p_{gt}^{i} - p_{pp}^{i}||$$

Where p_{gt}^{i} and p_{pp}^{i} are the i-th sampled point in the ground truth and proposed trajectories respectively. If AD is less than a fixed threshold, then the "good" label is assigned to the proposed trajectory and "bad" otherwise.

Furthermore, in order to refine the trajectories proposed during training, a parametrization is adopted for each trajectory as follows:

$$\begin{cases} t_x = x_e^{gt} - x_e^{pp}, \\ t_y = y_e^{gt} - y_e^{pp}, \\ t_\gamma = \gamma^{gt} - \gamma^{pp} \end{cases}$$

Where (x_e^{gt}, y_e^{gt}) are the coordinates of the ground truth endpoint, x_e^{pp}, y_e^{pp}, the coordinates of the endpoint of the proposed trajectory, and γ^{gt} and γ^{pp} the crooked angles of the ground truth and proposed trajectories respectively.

As for the loss function used, the authors employed a multi-task loss for this problem:

$$L = L_{ep}(p_e, p_e^*) + \frac{1}{N} \sum_{i} L_{cls}(c_i, c_i^*) + \frac{\alpha \sum_i L_{ref}(t_i, t_i^*)}{N_{pos} + \beta N_{neg}}$$

Notations used in the loss function:

$$
\begin{aligned}
p_e &- \text{proposed endpoint} \\
p_e^* &- \text{ground truth endpoint} \\
L_{ep} \text{ and } L_{ref} &- \text{L2 norm} \\
L_{cls} &- \text{cross entropy loss} \\
c_i &- \text{predicted label} \\
c_i^* &- \text{ground truth label} \\
t_i &- \text{parameters of the predicted trajectories} \\
t_i^* &- \text{parameters of the ground truth trajectory} \\
N_{pos} &- \text{number of positive labels} \\
N_{neg} &- \text{number of negative labels} \\
\alpha, \beta &- \text{hyperparameters}
\end{aligned}
$$

Metric	Dataset	S-LSTM [1]	S-GAN [12]	Liang [30]	Li [29]	SoPhie [38]	STGAT [17]	TPNet-1	TPNet-20
ADE	**ETH**	0.73 / 1.09	0.61 / 0.81	- / 0.73	- / **0.59**	- / 0.70	0.56 / 0.65	0.72 / 1.00	**0.54** / 0.84
	HOTEL	0.49 / 0.79	0.48 / 0.72	- / 0.30	- / 0.46	- / 0.76	0.27 / 0.35	0.26 / 0.31	**0.19** / **0.24**
	UNIV	0.41 / 0.67	0.36 / 0.60	- / 0.60	- / 0.51	- / 0.54	0.32 / 0.52	0.34 / 0.55	**0.24** / **0.42**
	ZARA1	0.27 / 0.47	0.21 / 0.34	- / 0.38	- / **0.22**	- / 0.30	0.21 / 0.34	0.26 / 0.46	**0.19** / 0.33
	ZARA2	0.33 / 0.56	0.27 / 0.42	- / 0.31	- / **0.23**	- / 0.38	0.20 / 0.29	0.21 / 0.33	**0.16** / 0.26
AVG		0.45 / 0.72	0.39 / 0.58	- / 0.46	- / **0.40**	- / 0.54	0.31 / 0.43	0.36 / 0.53	**0.27** / **0.42**
FDE	**ETH**	1.48 / 2.35	1.22 / 1.52	- / 1.65	- / 1.30	- / 1.43	**1.10** / **1.12**	1.39 / 2.01	1.12 / 1.73
	HOTEL	1.01 / 1.76	0.95 / 1.61	- / 0.59	- / 0.83	- / 1.67	0.50 / 0.66	0.48 / 0.58	**0.37** / **0.46**
	UNIV	0.84 / 1.40	0.75 / 1.26	- / 1.27	- / 1.27	- / 1.24	0.66 / 1.10	0.68 / 1.15	**0.53** / **0.94**
	ZARA1	0.56 / 1.00	0.42 / 0.69	- / 0.81	- / **0.49**	- / 0.63	0.42 / 0.69	0.55 / 0.99	**0.41** / 0.75
	ZARA2	0.70 / 1.17	0.54 / 0.84	- / 0.68	- / **0.55**	- / 0.78	0.40 / 0.60	0.43 / 0.72	**0.36** / 0.60
AVG		0.91 / 1.52	0.78 / 1.18	- / 1.00	- / 0.89	- / 1.15	0.62 / **0.83**	0.71 / 1.08	**0.56** / 0.90

Figure 2.11: Comparison of the TPNet implementations against other solutions, for a variety of datasets

2.6 Cross-domain Generalization for Motion Prediction

2.6.1 What is Cross-domain Generalization

Cross-domain generalization refers to the fact that the training and test distributions might present significant differences. For example, if the training and testing data have been sampled from two different countries, then environmental conditions, the behavior of drivers, the weather might differ significantly.

This paper by Ching-Yu Tseng, Po-Shao Lin et al. [17] tries to address this exact problem by bringing a self-attention mechanism and a new loss function to existing benchmarks. It is worth mentioning that this approach managed to obtain 3^{rd} place at the Shifts Challenge: Robustness and Uncertainty under Real-World Distributional Shift, competition held by NeurIPS in 2021.

2.6.2 Proposed model architecture

Similar to the idea mentioned in MotionCNN [11], the input to the model is going to be a multi-channel raster that encapsulates the historical data of the target agents, surrounding agents and the environment information.

The model uses NFNet [1] as backbone and a self-attention mechanism which aggregates groups of pixel-wise data. Using the extracted features, these are passed through a GRU layer [2], to recurrently generate the trajectories. An overview of the architecture can be seen in Fig. 2.12.

The loss function proposed is a combination of the negative log-likelihood, average distance error, and final distance error, summed up in order to minimize all three at the same time. Mathematically the loss used is defined as:

$$Loss = \sum_{Y} -log(p(Y;\theta)) + \sum_{l=1}^{T} (\hat{Y}_l - Y_l)^2 + (\hat{Y}_f - Y_f)^2$$

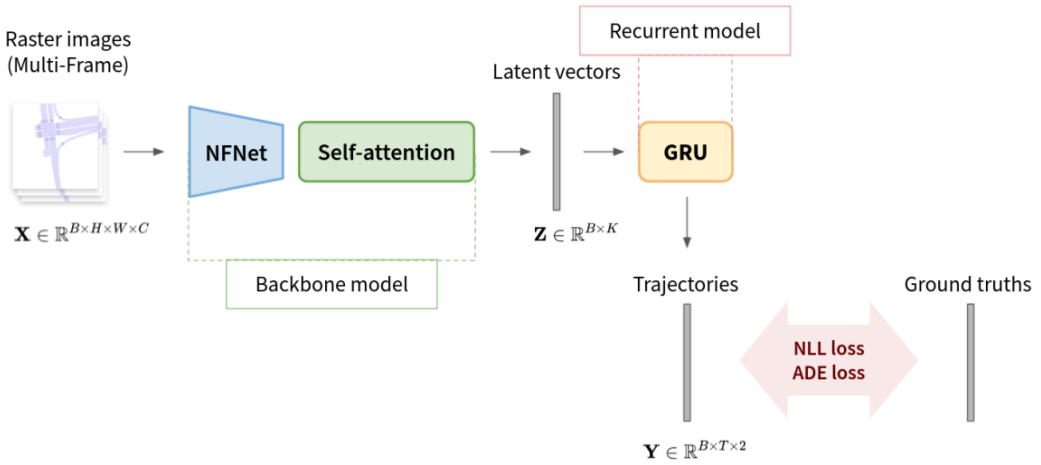

Figure 2.12: The architecture that obtained 3rd place in the Shifts Challenge: Robustness and Uncertainty under Real-World Distributional Shift, competition held by NeurIPS in 2021

Notations used in the loss function:

θ – the model parameters

$p(Y;\theta)$ – probability of trajectory Y being predicted conditioned to θ

Y_f – the final position in the trajectory Y

Rank	Method	Score (R-AUC CNLL)	CNLL↓	WADE↓	WFDE↓	MINADE↓	MINFDE↓
-	baseline	10.572	65.147	1.082	2.382	0.824	1.764
1	SBteam	**2.571**	15.676	1.850	4.433	0.526	1.016
2	Alexey & Dmitry	2.619	**15.599**	1.326	3.158	**0.495**	**0.936**
3	Ours	8.637	61.864	**1.017**	**2.264**	0.799	1.719

Figure 2.13: The results obtained by the methods that ranked in the first 3 places at the Shifts Challenge: Robustness and Uncertainty under Real-World Distributional Shift, and the baseline proposed for the competition. The metrics WADE, WFDE, and CNLL refer to the weighted variants of ADE, FDE, and NLL

Chapter 3

Qualitative results analysis

3.1 CARLA Simulator

3.1.1 Introduction

CARLA simulator [5] is an application specifically developed for research in the sphere of self-driving cars. The simulator has been built upon Unreal Engine 4 and represents an ideal environment for modeling autonomous vehicles, as it comes with multiple important aspects such as close to reality physics, pre-made maps, and full control over the world.

3.1.2 Features of the CARLA simulator

CARLA comes with some interesting features that really facilitate the development of autonomous agents. I am going to describe a few of them in the rest of this section.

Python API

The Python API exposed by the simulator is a really facile way of interacting with the actual simulation. It enables the developer to interact with the world in any way he could imagine, from spawning agents, to changing the state of traffic lights and to deep reinforcement learning training.

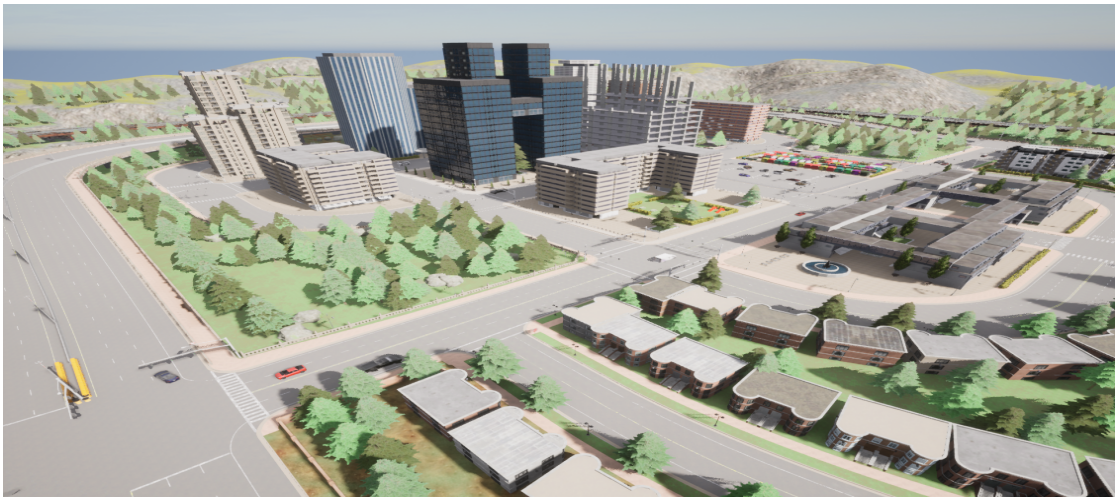

Figure 3.1: An example of a scene in the CARLA simulator

Scalability

By promoting a server multi-client architecture, CARLA brings an easy way of scaling an application and creating incredibly complex simulations that are as close to reality as possible. By default, the server runs on port 2000 and the clients from ports 2001 and on. This aspect can be easily reconfigured using the Python API.

Structured way of generating new maps

The maps used in the CARLA simulator are made under the OpenDRIVE [16] specification. This allows researchers to either create new maps manually or procedurally.

Sensors integration

CARLA comes with various built-in sensors like GNSS sensor, IMU sensor, etc. These can be really useful when testing a perception module or evaluating a fully autonomous driving car.

No render mode

To avoid additional overhead on the GPU that might come with rendering the simulation, CARLA provides a way of running it without rendering the actual graphics of the simulation, freeing this way resources that can be used for online training.

3.2 Implementation details for qualitative testing

3.2.1 Deep Learning models used

A preliminary implementation of the ReCoAt framework presented in the previous chapter shows that it is pretty slow on common GPUs. Processing a single scenario takes somewhere around 12 seconds on a GeForce GTX 1070. It is worth mentioning that the overhead includes an I/O operation of reading a large TFRecord file which compresses around 100 scenarios. Still, considering the fact that during a simulation the inference time takes into consideration parsing local data from the environment, this approach would not be feasible.

To avoid this bottleneck, I am going to employ the MotionCNN architecture for my qualitative analysis, since it is much simpler and does not involve so many pre-processing steps. Konev Stepan, Brodt Kirill, and Sanakoyeu Artsiom [11] offer to the research community the weights of their fully trained model. The files containing the weights can be found in their Github repository [12]. In the "Release" section they have published the weights of the model trained with both backbones, ResNet-18 and Xception71 respectively.

This model requires a lot of information about the environment. The complete list of the expected features is the following:

1. The ID, type, x-y coordinates, speed, velocity yaw, bounding box yaw, and the validity of each agent in the proximity of the vehicle, including itself in the last second and at the current time step

2. The width and length of each agent in the scene at the current time step

3. The future x-y coordinates and validity of all agents in the scene, but these fields are going to be just padded with -1 since they are not relevant for testing purposes

4. The ID, type, x-y coordinates, and validity of the roads, lane markings, crosswalks, and stop signs in the scene

5. The traffic lights' state, validity, and the ID of the road a specific traffic light affects

The exact specification of every feature can be found on the official documentation of the Waymo Open Dataset for Motion Prediction 2021 contest [18].

3.2.2 Leveraging the CARLA Python API

One thing worth mentioning is that since the problem assumes that the input to the motion prediction module is already given by some perception module, I am going to omit the perception part which involves working with data from sensors, and pass the relevant information to the model, taken directly as raw data from the simulation.

In this section, I am going to present implementation details and some code snippets on how to parse the necessary information from the environment using the Python API [4] that comes with the CARLA simulator.

Traffic lights data

The MotionCNN model only takes into consideration the current states of the traffic lights. To give relevant information to the model, I only took the traffic lights that affect roads that are in the proximity of 50 meters of the target agent.

```
traffic_lights = world_map.get_all_landmarks_of_type("1000001")
traffic_lights = [world.get_traffic_light(traffic_light) for
↪  traffic_light in traffic_lights]

for traffic_light in traffic_lights:
    state = traffic_light.state # the state of the traffic light
    for affected_road in
    ↪  traffic_light.get_affected_lane_waypoints(): # go through
    ↪  the list of the roads the traffic light affects
        road_id = affected_road.road_id # the id of the road
```

The state of an agent

The model requires both historical and current information about an agent. Even though there are is a lot of data that needs to be associated with one agent, the CARLA Python API, makes it really easy to extract the necessary features of one.

```python
# id of the agent
agent_id = agent.id

# current agent location
location = agent.get_transform().location
current_x, current_y = location.x, location.y

# width and length of the agent
current_width = np.array([[2 * agent.bounding_box.extent.x]])
current_length = np.array([[2 * agent.bounding_box.extent.y]])

# bounding box yaw
bbox_yaw = math.pi * agent.get_transform().rotation.yaw / 180 # degrees to radians

# velocity yaw
velocity = agent.get_velocity()
vel_yaw = math.atan2(velocity.y, velocity.x) # angle of the velocity vector

# new speed
location = np.array([location.x, location.y])
prev_location = np.array([lastX, lastY])

current_speed = np.linalg.norm(location - prev_location) / SAMPLING_RATE
```

Where SAMPLING_RATE is the rate at which new data must be fed to the model, in this case, 0.1 seconds, and the speed of the agent is calculated as:

$$s = \frac{\Delta d}{\Delta t}$$

Road information

The Python API provides a really simple way of getting road information out of the simulation. First of all, you need to get a list of all the waypoints on the map, sampled at a specific rate, in this case, 0.5 meters. Having these waypoints, you can then get the information about the lane markings next to them, if there are any. Also, the API gives the developers access to the crosswalks and stop signs, by simply calling a get method. The types are artificially enforced according to the Waymo Open Dataset for Motion Prediction documentation.

```
waypoints = world_map.generate_waypoints(
distance=DISTANCE_BETWEEN_WAYPOINTS)
for waypoint in self.roadgraph:
    if waypoint.lane_type is carla.LaneType.Driving:
        lane_location = waypoint.transform.location # location of a
        ↪  drivable lane

        right_vector = waypoint.transform.get_right_vector()

        # lane markings locations
        right_marking_location = lane_location + waypoint.lane_width
        ↪  / 2 * right_vector
        left_marking_location = lane_location - waypoint.lane_width
        ↪  / 2 * right_vector

        # lane markings types
        right_marking_type = waypoint.right_lane_marking.type
        left_marking_type = waypoint.left_lane_marking.type

        # lane markings colors
```

```
        right_marking_color = waypoint.right_lane_marking.color
        left_marking_color = waypoint.left_lane_marking.color

crosswalk_locations = world_map.get_crosswalks() # list of polylines
↪  representing the crosswalks

stop_signs = self.world_map.get_all_landmarks_of_type("206")
stop_signs = [world.get_traffic_sign(landmark) for landmark in
↪  stop_signs]
for stop_sign in stop_signs:
    stop_sign_location = stop_sign.get_location() # the location of
    ↪  the stop sign
```

Following the trajectory

After parsing all this data, it is passed to the MotionCNN model, which outputs 6 or 8 trajectories, depending on the backbone used. For ResNet-18 it is 6, while for Xception71 it is 8. I always use the trajectory with the highest confidence. To allow an agent to actually follow the trajectory, I used a PID controller [13], which is also included in the CARLA Python API.

3.3 Qualitative results

In the following photos, you can see on the left the behavior of the model with backbone ResNet-18, while on the right the one with Xception71.

Long term vs short term prediction

The first interesting result is the fact that the model with ResNet-18 backbone approximates long-term predictions better than the one with Xception71, as shown in Fig. 3.2. Although, not even the ResNet-18 one does not approximate well most of the time.

Considering this, I am going to use a shorter trajectory of 2 seconds for the rest of the experiments, because it yields better results overall, and also, this allows the agents to correct their mistakes a lot easier.

Figure 3.2: Comparison of the two backbones, on long-term prediction. The blue dots represent the lane markings the agents see

Figure 3.3: Comparison of the two backbones, on short-term prediction. The blue dots represent the lane markings the agents see

Interaction with other agents

Unfortunately, both models perform really badly when interacting with other agents. As shown in Fig. 3.4, none of the models try to even reduce their speed when approaching another agent, so they just end up crashing.

Figure 3.4: This figure shows the poor performance of the models when interacting with other agents

Junctions

When talking about junctions, it is worth mentioning that both models perform really well in most junctions because of the short-term predictions. We can see in Fig. 3.5 that both models can take a right turn with no problem.

Figure 3.5: Both models handle junctions pretty well when considering short-term prediction

In contrast, there are certain junctions that have weird markings that give trouble to both models, like the one in Fig. 3.6.

Figure 3.6: In some junctions with weird markings over them, both models perform really bad

Traffic lights

Other pretty bad results are shown in Fig. 3.7. In this case, neither model wants to stop at the red light. In this experiment, both cars move at a pretty high speed, around 30 km/h, so one reason they don't stop could be the fact that a sudden drop in speed can be dangerous for the passengers. Although, passing over a red light is not safe either.

Figure 3.7: Experiment involving traffic lights and high speed car

To combat this problem, I lowered the speed to 15 km/h and the Xception71 seems to be right in its prediction. The red dots in the right photo in Fig. 3.8, show that the model with the Xception71 backbone tries to predict the trajectory backward, indicating that the car should stop.

Figure 3.8: Experiment involving traffic lights and low speed car

Inference time

We can see from Fig. 3.9, that inference time is highly dependent on the number of surrounding agents. An overhead of 1.2 seconds for 20 agents on an Nvidia GTX 1070 is fairly high considering that the model should make these predictions every 2 seconds. It might be worth experimenting with models that do not require so much preprocessing like DenseTNT[8].

Number of agents	ResNet-18	Xception71
1	0.1 seconds	0.12 seconds
10	0.6 seconds	0.6 seconds
20	1.2 seconds	1.2 seconds

Figure 3.9: Inference time of both models based on the number of surrounding agents

Chapter 4

Future work

4.1 Training bottleneck

A big bottleneck that I faced during my research was training the deep learning models using the Waymo dataset. This dataset is really large and contains roughly 600GB of comprised data.

The lack of specialized hardware stopped me from experimenting more with new ideas of improving the models and combining different ideas to see how they work. To give an approximation of the hardware needed, besides disk data that should be enough to store the entire dataset, the team that developed the DenseTNT model [8] used 8 GeForce RTX 2080Ti GPUs in parallel during training according to this Issue on their Github repository [15] and it still took them around 2 days to train it only once.

Instead of investing in specialized hardware like the Nvidia Tesla V100 GPU, which is a GPU created specifically for AI training, a more affordable solution would be to make use of cloud resources, from providers like AWS or Azure. This would drive down the costs, but it would require setting up CI/CD pipelines for deploying the models in the cloud and training them there.

4.2 Further experimentation

After figuring out the hardware problem, I would like to further experiment with the other 4 ideas I have presented in this thesis. Furthermore, I think that investing time in researching more ideas, and coming up with new solutions from my research would bring a pretty big benefit towards developing a robust solution for this problem.

Finding a good trade-off between complexity and performance on these kinds of deep learning models could prove to be a stepping stone towards fully autonomous vehicles, as this problem is still under development. In my opinion, good generalization capability and time performance is even more important than collecting additional data, as we already have a lot of datasets that contain scenarios from real-world data.

4.3 Beyond self-driving cars

The ideas presented in this thesis are not only tied to fully autonomous vehicles. They can be extended to other industries as well.

The gaming industry for example can benefit a lot from implementing these ideas into their projects. Instead of having the developers hard-code rules for their cars, which can take a lot of time, they could gather the information from their game, use the output of one of the models presented in this thesis and make the car follow that specific trajectory. This has the potential of freeing a lot of time for the developers who could focus on more important tasks this way, which in turn would drive the development costs down for the organization.

Bibliography

[1] Andrew Brock, Soham De, Samuel L. Smith, and Karen Simonyan. "High-Performance Large-Scale Image Recognition Without Normalization." In: *CoRR* abs/2102.06171 (2021). arXiv: 2102.06171. URL: https://arxiv.org/abs/2102.06171.

[2] KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches." In: *CoRR* abs/1409.1259 (2014). arXiv: 1409.1259. URL: http://arxiv.org/abs/1409.1259.

[3] François Chollet. "Xception: Deep Learning with Depthwise Separable Convolutions." In: *CoRR* abs/1610.02357 (2016). arXiv: 1610.02357. URL: http://arxiv.org/abs/1610.02357.

[4] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. "CARLA simulator Python API documentation page." In: (). URL: https://carla.readthedocs.io/en/0.9.13/python_api/.

[5] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. "CARLA: An Open Urban Driving Simulator." In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[6] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. "The pascal visual object classes (voc) challenge." In: *International journal of computer vision* 88.2 (2010), pp. 303–338.

[7] Liangji Fang, Qinhong Jiang, Jianping Shi, and Bolei Zhou. "TPNet: Trajectory Proposal Network for Motion Prediction." In: *CoRR* abs/2004.12255 (2020). arXiv: 2004.12255. URL: https://arxiv.org/abs/2004.12255.

[8] Junru Gu, Chen Sun, and Hang Zhao. "Densetnt: End-to-end trajectory prediction from dense goal sets." In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 15303–15312.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition." In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[10] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory." In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[11] Stepan Konev, Kirill Brodt, and Artsiom Sanakoyeu. "MotionCNN: A Strong Baseline for Motion Prediction in Autonomous Driving." In: (2021).

[12] Stepan Konev, Kirill Brodt, and Artsiom Sanakoyeu. "Waymo motion prediction challenge 2021: 3rd place solution." In: 2021. URL: https://github.com/kbrodt/waymo-motion-prediction-2021/tree/main.

[13] Melder Nic and Tomlinson Simon. "Racing Vehicle Control Systems using PID Controllers." In: (). URL: http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter40_Racing_Vehicle_Control_Systems_using_PID_Controllers.pdf.

[14] Shahrokh Paravarzar and Belqes Mohammad. "Motion Prediction on Self-driving Cars: A Review." In: *CoRR* abs/2011.03635 (2020). arXiv: 2011.03635. URL: https://arxiv.org/abs/2011.03635.

[15] ShoufaChen. "Training time on Waymo. Issue on the official DenseTNT Github repository." In: Apr. 2022. URL: https://github.com/Tsinghua-MARS-Lab/DenseTNT/issues/19.

[16] Daimler Driving Simulator Stuttgart and VIRES Simulationstechnologie GmbH. "OpenDRIVE specification." In: 2019. URL: https://www.asam.net/standards/detail/opendrive/.

[17] Ching-Yu Tseng, Po-Shao Lin, Yu-Jia Liou, Kuan-Chih Huang, and Winston H. Hsu. "3rd Place Solution for NeurIPS 2021 Shifts Challenge: Vehicle Motion Prediction." In: *CoRR* abs/2112.01348 (2021). arXiv: 2112.01348. URL: https://arxiv.org/abs/2112.01348.

[18] Waymo. "Waymo Open Dataset for Motion Prediction tf.Example Format." In: 2021. URL: https://waymo.com/open/data/motion/tfexample.

[19] Ronald J. Williams and David Zipser. "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks." In: *Neural Computation* 1.2 (1989), pp. 270–280. DOI: 10.1162/neco.1989.1.2.270.

[20] Chen Lv Zhiyu Huang Xiaoyu Mo. "ReCoAt: A Deep Learning Framework with Attention Mechanism for Multi-Modal Motion Prediction." In: (2021). URL: https://drive.google.com/file/d/1Ksq7X5dzouMV2jG1QYcgWzpUl2dKWUDW/view.