Vision-Centric Autonomous Driving

Hao Yang, *1753300, Robotics, Cognition and Intelligence, TUM*
guided by Liguo Zhou

***Abstract*—This article explores the feasibility and performance of pure visual autonomous driving. A paper from Shanghai AI Lab discusses the trajectory point prediction output by model ST-P3 after processing photos from cameras and data from IMU and radar sensors, with L2 value and collision rate as evaluation metrics. Referencing this paper, this article investigates the model's output results under two scenarios: pure coordinate input and joint input of coordinates and photos. Additionally, to reduce the L2 loss value, this article also attempts different approaches to process the output data and sample label values, ultimately achieving satisfactory results.**

***Index Terms*—input with image and coordinate, transformation of coordinates in different frames, LSTM, NuScenes, Loss optimization**

I. Introduction

AS autonomous driving technology continues to advance, and the integration of an expanding array of sensors for data input has become increasingly prevalent. However, the incorporation of additional sensor inputs inevitably results in a surge of data that demands substantial computational resources for real-time processing. Notably, Tesla's Autopilot system relies exclusively on cameras as its primary sensor type. While it faces challenges in adverse weather conditions, it has demonstrated impressive performance under normal weather conditions. This highlights the potential viability of using cameras as the sole input sensor, prompting a deeper exploration of this approach.

This research project aims to design an autonomous driving model that relies exclusively on camera input sensors. The objective is to process incoming images from the camera and employ machine learning-based path-planning techniques to predict the car's trajectory. These predicted trajectory points can then be converted into real-time signals to control crucial aspects of the vehicle, such as the gas pedal, brakes, and steering angle.

The chosen dataset for this project is the NuScenes dataset, and the results will be compared with the results of a recent paper [1] from Shanghai AI Lab. The project is divided into two phases to comprehensively investigate the use of cameras as the primary sensor input.

In the initial phase, only the coordinates of the previous time are employed. The model has two kinds of architectures: one model architecture employs a single LSTM network, directly processing all past coordinates of the inputs and predicting the coordinates of the required future trajectory points in one go. Another model architecture is more complex, involving multiple iterations of LSTM networks. In each iteration, a fixed number of past coordinate points are inputted, followed by the prediction of a future trajectory point. This process is repeated multiple times, ultimately yielding the coordinates of all the trajectory points that need to be predicted.

Subsequently, in the second phase, the front-facing cameras and the coordinates of the vehicle are employed together. The model consists of two components. One component aligns with the network that processes coordinates in the first stage, while the other employs a ResNet network to handle images. Finally, the fully connected layers obtained from processing the two components are concatenated together and undergo simple processing to obtain the final predicted trajectory points.

Due to the large L2 loss incurred by directly computing the trajectory point's loss for the entire output sequence, in order to reduce the loss and improve the accuracy of training results, this paper draws inspiration from video coding techniques such as frame grouping and inter-frame compression. We have redefined the processing of model prediction values. By calculating the distance between adjacent trajectory points predicted rather than between each trajectory point and the current position, and using this to compute the loss, we have successfully improved the prediction accuracy of the model. This has resulted in a significant reduction in both the loss value and collision rate.

The network architecture for image and coordinate processing in this paper is not overly complex, merely involving a simple application of ResNet and LSTM structures. Comparing the predicted results with those of the STP3 network, our model exhibits superior performance. However, both predictions still require further optimization. Future endeavors may involve experimenting with more sophisticated network models, such as transformers, to better extract corresponding features and achieve more accurate predictions.

The paper begins with an introduction, followed by an explanation of the data preparation measures taken in this project. Next, the applied models and estimation methods are introduced. Subsequently, problems encountered during the training process and their underlying causes are described. Furthermore, the test results of various models and estimation criteria are elucidated. Finally, the conclusion and discussion of this project are presented.

II. Data Preparation

In the initial phase of the project, raw data from the NuScenes Dataset undergoes essential preprocessing steps. This includes applying provided coordinates and rotation matrices to ensure consistency in the coordinate system across different images. Furthermore, as the given images are of size 1900 * 600, resizing and augmentation become imperative. Resizing not only standardizes the image dimensions but also accelerates the training process by reducing computational

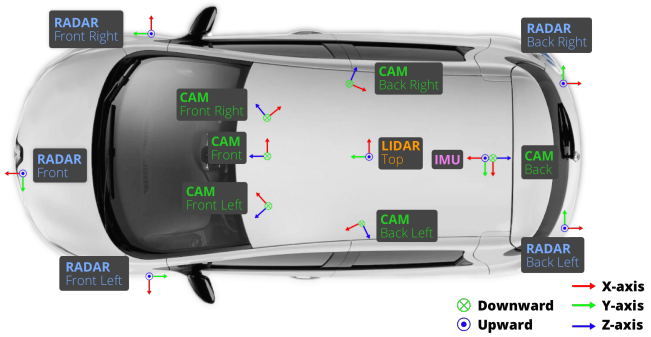

Fig. 1: Car Setup [2]

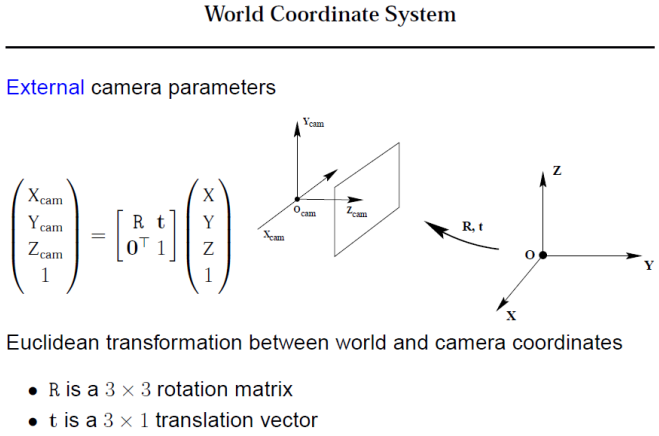

Fig. 2: Coordinate System Transformation [3]

overhead. Augmentation techniques enhance the diversity and robustness of the training dataset, ensuring better generalization of the model. These measures are crucial in streamlining the training pipeline and accommodating large-scale datasets effectively

A. Transformation of coordinate-frames

To capture real-world scenes in a photograph, a crucial step involves transforming coordinates from the world coordinate system to the camera coordinate system. Furthermore, it's essential to adjust the coordinate system to align with the working vehicle's frame, which requires predicting its trajectory. This transformation necessitates accounting for the camera's position and orientation relative to the world. Once the camera coordinates are established, the subsequent step involves projecting these coordinates onto the image plane to derive image coordinates. This projection process incorporates parameters such as focal length, principal point, and distortion coefficients, ensuring accurate mapping of points from the camera coordinate system to their corresponding locations in the 2D image. Through this transformation, a geometric relationship between the three-dimensional world and the two-dimensional image is established, facilitating the representation of complex scenes in a photograph.

Within the dataset, both a position matrix based on the world coordinate system and a rotation matrix in quaternion form are provided. To undertake the transformation tasks, the initial step involves constructing a transformation matrix. This task is accomplished by following the provided pseudo-code algorithm 1. Then we can get global pose coordinates in the same coordinate system by following the pseudo-code of algorithm 2.

Algorithm 1 Transform Matrix Computation

Input:
translation: np.array($[0, 0, 0]$),
rotation: Quaternion($[1, 0, 0, 0]$),
inverse: bool = False
Output:
tm: np.ndarray = np.eye(4)

tm $\leftarrow$ np.eye(4);

if *inverse* **then**
    rot_inv $\leftarrow$ rotation.rotation_matrix.T;

    trans $\leftarrow$ -np.array(translation).T;

    tm$[: 3, : 3] \leftarrow$ rot_inv;

    tm$[: 3, 3] \leftarrow$ rot_inv.dot(trans);
end
else
    tm$[: 3, : 3] \leftarrow$ rotation.rotation_matrix;

    tm$[: 3, 3] \leftarrow$ np.array(translation).T;
end

return *tm*;

B. Pre-process of input images

Algorithm 2 Calculate global pose transformation

Input:
ep_t: translation vector of ego pose
ep_r: rotation quaternion of ego pose
cs_t: translation vector of sensor pose
cs_r: rotation quaternion of sensor pose
$inverse$: flag indicating whether to compute the inverse transformation
Output:
$pose$: transformation matrix representing the global pose

if $inverse$ *is* ***false*** **then**
  $global\_from\_ego \leftarrow$ transform_matrix(ep_t, Quaternion(ep_r), inverse=False)
  $ego\_from\_sensor \leftarrow$ transform_matrix(cs_t, Quaternion(cs_r), inverse=False)
  $pose \leftarrow global\_from\_ego \cdot ego\_from\_sensor$
end
else
  $sensor\_from\_ego \leftarrow$ transform_matrix(cs_t, Quaternion(cs_r), inverse=True)
  $ego\_from\_global \leftarrow$ transform_matrix(ep_t, Quaternion(ep_r), inverse=True)
  $pose \leftarrow sensor\_from\_ego \cdot ego\_from\_global$
end
return $pose$

In this project, a series of consecutive four-frame images along with their corresponding car coordinates are inputted into the processing model for training. The images obtained directly from the dataset have a resolution of $900 * 1600$, which is considerably large, especially when considering the combined size of the four frames. To enhance the training speed while maintaining effective feature extraction from the images, preprocessing is necessary. Firstly, the image resolution needs to be resized to a smaller size, such as $270 * 480$. Additionally, a random cropping process is applied to further reduce the resolution. Moreover, normalization is performed on the images.

Algorithm 3 Get Resizing and Cropping Parameters

Input:
Original image dimensions:
$original\_height$, $original\_width$
Final desired dimensions: $final\_height$, $final\_width$
Resize scale: $resize\_scale$
Crop height: $crop\_h$

Output: Resizing and cropping parameters

$resize\_dims \leftarrow (int(original\_width \times resize\_scale), int(original\_height \times resize\_scale))$
$resized\_width, resized\_height \leftarrow resize\_dims$
$crop\_w \leftarrow int(max(0, (resized\_width - final\_width)/2))$
$crop \leftarrow (crop\_w, crop\_h, crop\_w + final\_width, crop\_h + final\_height)$
if $resized\_width \neq final\_width$ **then**
  Print "Zero padding left and right parts of the image."
end
if $crop\_h + final\_height \neq resized\_height$ **then**
  Print "Zero padding-bottom part of the image."
end
return *Resizing and cropping parameters:*
$\{scale\_width : resize\_scale, scale\_height : resize\_scale, resize\_dims : resize\_dims, crop : crop\}$

In this project, the resizing and cropping parameters are firstly obtained by pseudo-code of algorithm 3. The imported images are then resized and cropped according to the obtained parameters. Finally, the resulting images undergo normalization before being converted into a tensor format suitable for GPU processing. With these steps completed, the image processing pipeline is finalized.

The function, "normalize," plays a crucial role in image processing by transforming the coordinates of image points. It applies a two-step normalization process to the input coordinates. First, it utilizes the sigmoid function to squash the values between 0 and 1, effectively enhancing the contrast and ensuring the output lies within a bounded range. Subsequently, it linearly scales the values to span the range [-1, 1], providing a standardized coordinate system suitable for various image manipulation tasks. This normalization facilitates consistent and efficient processing of image data, enabling better performance in tasks such as object detection, image registration, and feature extraction. Here is the pseudo-code of the normalization function 4.

Methods	L2(m)			Collison(%)		
	1s	2s	3s	1s	2s	3s
Vanilla	0.50	1.25	2.80	0.68	0.98	2.76
NMP [5]	0.61	1.44	3.18	0.66	0.90	2.34
Freespace [4]	0.56	1.27	3.08	0.65	0.86	1.64
ST-P3	1.33	2.11	2.90	0.23	0.62	1.27

TABLE I: Open-loop planning results [1]

Algorithm 4 Normalize Coordinates

Data:
Coordinates $coors$
Result:
Normalized coordinates

$coors \leftarrow 1/(1 + e^{-coors})$
$coors \leftarrow 2 * coors - 1$
return $coors$

III. Different Models and Loss Calculation

The project explores end-to-end autonomous driving solely relying on visual inputs, drawing inspiration from the STP3 project's research paper [1]. The paper outlines two approaches: one that inputs coordinates only and another that simultaneously processes coordinates and images. Evaluation metrics include L2 distance and collision rate. The trajectory prediction results can be seen in table **??**. In the STP3 project, a Bird's Eye View (BEV) solution was employed to construct the neural network. In this project, we attempt to utilize a ResNet50 network for image processing and feature extraction, while employing an LSTM network to handle sequential coordinate data. Subsequently, we dynamically adjust the fusion ratio between image and coordinate features as needed, ultimately outputting predicted coordinates for trajectory points. The following sections provide detailed explanations of the various network structures utilized in this project.

A. Resnet50

ResNet, short for Residual Networks, stands as one of the most widely acclaimed Convolutional Neural Network (CNN) architectures. Its groundbreaking introduction in 2015 by Microsoft Research, as detailed in the paper authored by He et al., shattered numerous records in the field of deep learning. As shown in picture 4, the ResNet-50 architecture can be broken down into 6 parts:

1) *Input Pre-processing*
   Before images are inputted into the model, we perform some data augmentation to allow our model to be more robust. A random flip, random rotation, random contrast, random shadow, random brightness, and some Gaussian noise are used to make the image set more varied. The parameters of the functions are probabilities, i.e. the chance that an image will undergo the selected transformation.
2) *Cfg[0] blocks*
   This block comprises a convolutional layer followed by two Identity layers. To enhance numerical stability, a kernel constraint enforces weight normalization at

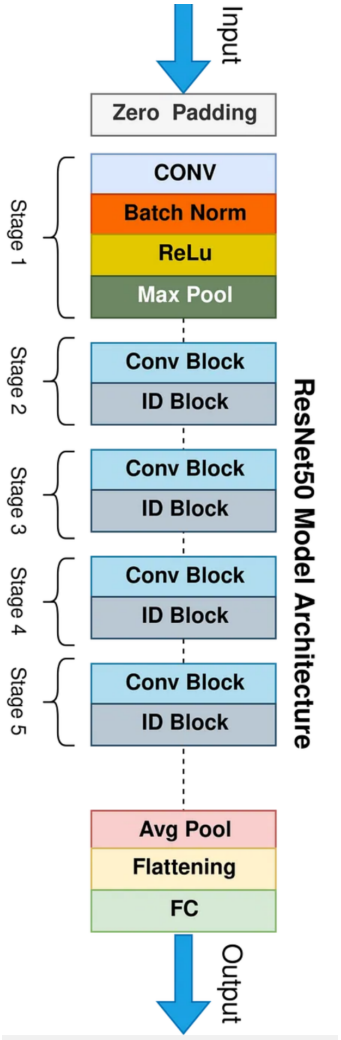

Fig. 3: Structure of Resnet50 [6]

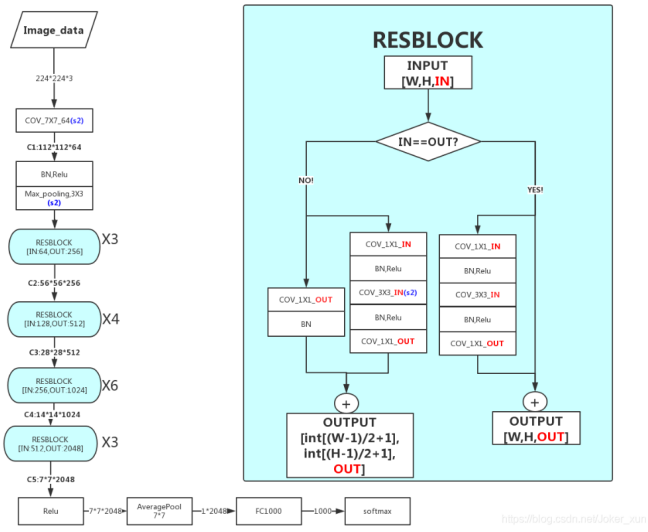

Fig. 4: Architecture of Resnet50 [8]

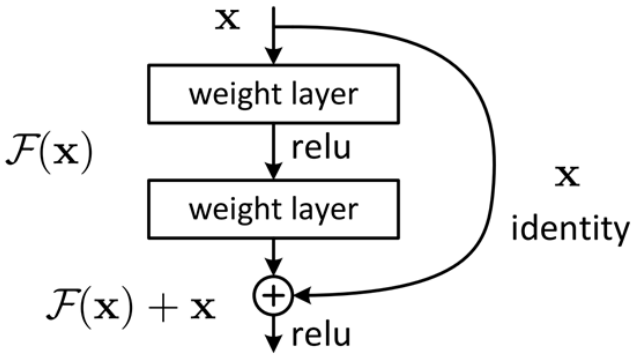

Fig. 5: Definition of residual block [7]

regular intervals. Batch Normalization layers are inserted between each pair of consecutive layers. As shown in the picture 6, the identity layer means that the output is unchanged from the input of the function.
Input Shape : (56,56,64)
Output Shape : (56,56,256)

3) *Cfg[1] blocks*
This block consists of one Convolutional Layer and two Identity Layers, resembling the structure of the Cfg0 blocks. The primary distinction lies in the increased number of out channels for both the Convolutional and Identity layers.
Input Shape : (56,56,256)
Output Shape : (28,28,512)

4) *Cfg[2] blocks*
This block consists of 1 Convolutional layer followed by 5 Identity layers. It holds particular significance within the ResNet architecture, as variations of the model often diverge in this block.
Input Shape : (28,28,512)
Output Shape : (14,14,1024)

5) *Cfg[3] blocks*
In this network, the final set of Convolutional Layer blocks consists of 1 Convolutional Layer followed by 2 Identity Layers.
Input Shape : (14,14,1024)
Output Shape : (7,7,2048)

6) *Fully-Connected layer*
In this block, we employ an Average Pooling Layer, a Dropout Layer, and a Flatten Layer. Subsequently, the feature map undergoes flattening and is passed into a Fully Connected Layer, which serves as the foundation for producing predictive outcomes. A Softmax activation function is applied to yield logits and probabilities.
In the initial task of our project, the sole objective entails generating a steering angle output. Consequently, the output channel has been configured to a single channel. However, in the subsequent task, we necessitate the generation of three distinct signals: throttle, brake, and steering angle. To facilitate this, we have set the parameter "CLASS TYPE" (or output channel) to three, aligning with the multifaceted nature of the second task as outlined in our research paper.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Fig. 6: Architecture of Resnet series [7]

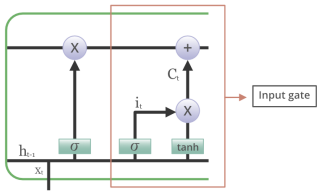

Fig. 7: input gate [10]

Input Shape : (7,7,2048)
Output Shape : (1, CLASS TYPES)

In the ResNet architecture, different versions utilize varying quantities of configuration blocks denoted as 'Cfg blocks' at different hierarchical levels, as illustrated in the accompanying figure 6. A comprehensive breakdown of these configurations is presented below to provide a thorough understanding of the network's architectural variations.

In this project, a pre-trained ResNet50 model is utilized, and the final fully connected (fc) layer is modified to output either a 512-dimensional or a 12-dimensional fc layer. When outputting a 512-dimensional fc layer, it is concatenated with the fc layer from the coordinate processing model at a 1:1 ratio. When outputting a 12-dimensional fc layer, it is concatenated with the fc layer from the coordinate processing model at a 12:96 ratio. The different ratio is used to find a better combination of feature extraction results from the image and coordinate process model.

B. LSTM

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) architecture specifically designed to address the vanishing gradient problem commonly encountered in traditional RNNs. Unlike standard RNNs, LSTM networks utilize a more complex structure composed of multiple interconnected memory cells, which enables them to effectively capture long-range dependencies in sequential data.

At the core of an LSTM network are memory cells, each equipped with three main components: a cell state, an input gate, and an output gate.

The cell state serves as a conveyor belt, transporting information across time steps.

The input gate plays a crucial role in adding valuable information to the cell state. Initially, information undergoes regulation via the sigmoid function, filtering the values to be

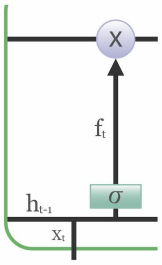

Fig. 8: forget gate[10]

retained similar to the forget gate, utilizing inputs ht-1 and xt. Subsequently, a vector is generated using the tanh function, producing an output ranging from -1 to +1, encompassing all potential values from ht-1 and xt. Finally, the values of the vector and the regulated values are multiplied to derive useful information. The equation for the input gate is as follows:

$$i_t = (W_i[h_{t-1}, x_t] + b_i)$$
$$\hat{C}_t = tanh(W_c[h_{t-1}, x_t] + b_c)$$
$$C_t = f_t C_{t-1} + i_t \hat{C}_t$$

The previous state is multiplied by f_t, incorporating the information that was previously selected to be disregarded. Next, $i_t * C_t$ is added, representing the revised candidate values adjusted based on the extent to which each state value was chosen to be updated. The symbol $\odot$ in Figure 7 signifies element-wise multiplication, while the function tanh denotes the hyperbolic tangent activation function.

Simultaneously, the forget gate is responsible for removing information that is no longer useful in the cell state. It takes two inputs, x_t (input at the current time step) and h_{t-1} (previous cell output), which are then multiplied by weight matrices and added to a bias term. The resulting value is passed through an activation function, yielding a binary output. If the output is 0 for a particular cell state, the corresponding information is forgotten; if it's 1, the information is retained for future use. The equation for the forget gate can be expressed as follows:

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

where: W_f represents the weight matrix associated with the forget gate. $[h_{t-1}, x_t]$ is the concatenation of the current input and the previous hidden state. b_f denotes the bias with the forget gate. σ represents the sigmoid activation function.

Finally, the output gate is responsible for extracting valuable information from the current cell state to be presented as output. Initially, a vector is created by applying the hyperbolic tangent (tanh) function to the cell state. This vector is then modulated using the sigmoid function to regulate the flow of information, filtering which values to retain based on inputs from the previous cell state (ht-1) and the current input (x_t). Finally, the vector values are multiplied by the regulated values and transmitted as both output and input to the next cell. The equation for the output gate is:

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

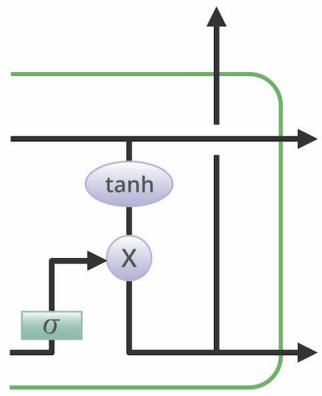

Fig. 9: output gate [10]

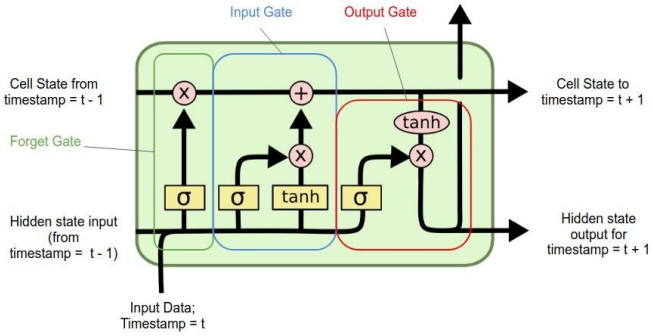

Fig. 10: Structure of LSTM [9]

$$\text{output} = \tanh(C_t) \odot \text{output gate}$$

Here, W_o and b_o represent the weight matrix and bias vector of the output gate, respectively. h_{t-1} and x_t denote the previous hidden state and current input, while C_t signifies the current cell state. The $\odot$ symbol denotes element-wise multiplication.

One of the key advantages of LSTM networks is their ability to learn and remember patterns over long sequences, making them well-suited for tasks involving sequential data, such as natural language processing, speech recognition, and time series forecasting. In our project, the input data is coordinate sequence and image sequence, and the predicted trajectory is also time series forecasting, which makes LSTM a very suitable model for this project. Additionally, LSTM networks are robust to the challenges of gradient vanishing and exploding, which commonly plague traditional RNNs, allowing them to maintain stable learning even over extended sequences.

In practice, LSTM networks have demonstrated impressive performance across a wide range of applications, from generating coherent text to accurately predicting future values in financial markets. Their versatility, coupled with their ability to model intricate temporal dependencies, has solidified their position as one of the go-to choices for sequence modeling tasks in the field of deep learning.

C. Purely coordinates processing model

Algorithm 5 one LSTM Network for coordinate process

Input:
Input sequence x with $x[1]$ as the coordinate tensor

Output:
Output feature tensor

Function initialization
$input\_n \leftarrow 4$
$out\_feature \leftarrow 12$
$in\_feature \leftarrow 2 \times input\_n - 2$
$lstm \leftarrow$ `CustomLSTM` (input_size=2, hidden_size=512, num_layers=2, num_classes=512)
$fc \leftarrow$ `Sequential(Linear(`*512, 100*`), ReLU(), BatchNorm1d(`*100*`), Linear(`*100, out_features=out_feature*`))`
End

Function forward(x)
$x\_coordinate \leftarrow x[1].reshape(-1, 3, 2)$
$x \leftarrow$ lstm($x\_coordinate$)
$output \leftarrow$ fc(x)
return *Output*
End

Algorithm 6 Coordinate Processing

Data:
old coordinates x_1,
predicted coordinates from LSTM x_2

Result:
Processed coordinates x

Function forward(x)
$x_0 \leftarrow$ concatenate$(x_1, x_2, \text{dim} = 1)$
$x \leftarrow x_0[:, 1:, :]$
return x
End

There are two types of network structures for dealing with pure coordinate data, both related to LSTM. The first involves directly feeding the coordinates of cars in four consecutive frames into an LSTM network and then predicting all 6 trajectory points' coordinates at once. The pseudo-code is shown in algorithm 5.

The second type entails inputting the coordinates of cars in the first four frames, predicting the coordinates of the trajectory points needed for the next frame, then feeding the predicted coordinates along with the trajectory points' coordinates from the previous three frames into the model to predict the coordinates for the next frame, iterating until all required trajectory points' coordinates are predicted. The pseudo-code is shown in algorithm 7.

The first network model is simpler, while the second is more complex but can reduce errors caused by longer-term predictions. Specific comparative results can be seen in subsequent tests.

Algorithm 7 Net_coo, multi_lstm

Input :
Input coordinates x

Output:
Transformed coordinates *output*

Initialize LSTM network with parameters

Function forward(x)
coo ← Extract coordinates from input x, (batch_size * 4 * 2)
$x_1 \leftarrow$ Apply LSTM to *coo*
coo ← Process coordinates using `process_coo` function with *coo* and x_1
$x_2 \leftarrow$ Apply LSTM to *coo*
coo ← Process coordinates using `process_coo` function with *coo* and x_2
$x_3 \leftarrow$ Apply LSTM to *coo*
coo ← Process coordinates using `process_coo` function with *coo* and x_3
$x_4 \leftarrow$ Apply LSTM to *coo*
coo ← Process coordinates using `process_coo` function with *coo* and x_4
$x_5 \leftarrow$ Apply LSTM to *coo*
coo ← Process coordinates using `process_coo` function with *coo* and x_5
$x_6 \leftarrow$ Apply LSTM to *coo*
output ← Concatenate x_1, x_2, x_3, x_4, x_5, x_6
output ← Reshape output to $(-1, 12)$
output ← Apply fully connected layers to output
return *Output*
End

D. Coordinates and images processing together model

From the results presented in Table I, it is evident that utilizing pure coordinates as input may yield a relatively low L2 loss. However, it can also lead to a higher collision rate, which is arguably more significant in practical applications. To mitigate the collision rate, images are incorporated as additional input in this project. Specifically, ResNet50 is employed to extract features from the images, which are then combined with the features obtained from the coordinate processing network. Analogous to the network architecture employed in the pure coordinate section, two different architectures are utilized in this section, corresponding to the coordinate-based approach.

Algorithm 8 Pseudo-code for the Net_img algorithm

Input :
x: Input data containing image and coordinates

Output:
$output$: Predicted output

Function Net_img(x)
Initialize LSTM network with input size 2 and hidden size 512
Initialize ResNet50Custom with output classes 12
Initialize fully connected layers: $fc1$ and fc
$coo \leftarrow x[1].reshape(-1, 3, 2)$
$x1 \leftarrow LSTM(coo)$
$x1 \leftarrow x1.reshape(-1, 1, 2)$
for $i \leftarrow 2$ **to** 6 **do**
  $coo \leftarrow$ process_coo(coo, x_i)
  $x_i \leftarrow$ LSTM(coo)
  $x_i \leftarrow x_i.reshape(-1, 1, 2)$
end
$output1 \leftarrow$ concatenate($x1, x2, x3, x4, x5, x6$) along dimension 1
$output1 \leftarrow output1.reshape(-1, 12)$
$output1 \leftarrow fc1(output1)$
$img \leftarrow x[0]$
$img \leftarrow$ ResNet50Custom(img)
$output \leftarrow$ concatenate($img, output1$) along dimension 1
$output \leftarrow fc(output)$
return $output$
End

In addition to different network structures based on coordinate grids, this project also categorizes the fusion of image features and coordinate features into two ratios: 1:1 and 12:96, depending on their respective proportions. If the features from resnet50 are useless or even worse, 12:96 ratio can help reduce the bad influence of image features. Otherwise, 1:1 ratio can promise that the impact of image features is not ignored by the model.

Algorithm 9 image and coordinate as input, one lstm

Input :
x: Input data

Output:
output: Output of the network

Data:
input_n: Number of input features,
out_feature: Number of output features

Function *MyNetwork(x, input_n, out_feature)*:
in_feature $\leftarrow 2 \times$ input_n $- 2$
img $\leftarrow$ `ResNet50Custom(`$x[0]$`)`
coo $\leftarrow x[1]$.reshape$(-1, 3, 2)$
coo $\leftarrow$ `CustomLSTM(`*coo, input_size=2, hidden_size=512, num_layers=2, num_classes=96*`)`
output $\leftarrow$ concatenate(img, coo)
output $\leftarrow$ fully_connected(output, $108, 48$)
output $\leftarrow$ ReLU(output)
output $\leftarrow$ fully_connected(output, 48, out_features)
return *output*

E. L1 and L2 calculation

When discussing loss functions in deep learning, two commonly mentioned ones are the L1 and L2 loss functions. They are utilized to evaluate the disparity between the model predictions and the actual targets, serving as optimization objectives during neural network training. Below, these two loss functions and their computation methods will be briefly introduced.

The L1 loss function measures the absolute discrepancy between the model's predictions and the actual targets. Its computation method is as follows:

$$L_1(y, \hat{y}) = \sum_{i=1}^{n} |y_i - \hat{y}_i|$$

Where y represents the actual targets, $\hat{y}$ denotes the model's predictions, and n is the number of samples. This means that for each sample, we calculate the absolute difference between its predicted value and the true value and sum up all the differences. The L1 loss function is more robust when handling outliers since it has lower sensitivity to them.

The L2 loss function measures the squared discrepancy between the model's predictions and the actual targets. Its computation method is as follows:

$$L_2(y, \hat{y}) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Unlike the L1 loss, the L2 loss is the sum of squared differences between the predicted values and the true values. This amplifies the impact of larger errors in the computation. In many cases, the L2 loss function results in a smoother optimization path but is more sensitive to outliers compared to L1 loss.

These two loss functions play crucial roles in training neural networks and can be chosen based on the specific task and data characteristics. Typically, when the task demands greater robustness against outliers, the L1 loss function is preferred. Conversely, if a smoother optimization path is desired, the L2 loss function is chosen. In practical applications, sometimes a combination of both loss functions is used to create a new loss function that considers multiple factors comprehensively.

F. modified L2 Loss

In the training process using L2 loss, it was observed that the resulting loss values were not ideal. One possible reason is related to the model's predictions. In this project, the model predicts the coordinates of a car in the next 3 seconds, which corresponds to 6 frames of images. As the prediction time point moves further away from the current frame, the error in the predicted coordinates also increases. Consequently, this leads to an increase in the loss value.

To mitigate the impact of this factor, a new prediction calculation method is employed in this project. When calculating the loss value, the predicted coordinates are transformed into the difference relative to the coordinates of the preceding frame. Similarly, the actual results are processed. Consequently, the loss calculation focuses on the coordinate differences between adjacent frames. This theoretically reduces the loss value significantly. The following pseudo-code (Algorithm 10) illustrates the specific operation:

Algorithm 10 Modified L2 Calculation

Input:
coor_past, image, model, optimizer, cfg.predict_n
Output:
loss
Forward
    delta_sum_xy ←
     torch.zeros((coor_past.size(0), 2)).to(device)
    optimizer.zero_grad()
    output ← model([image, coor_past])
    for $i \leftarrow 1$ **to** *cfg.predict_n* **do**
        delta_sum_xy[:, 0] ← delta_sum_xy[:, 0] + output[:, 2 * (i - 1)]
        delta_sum_xy[:, 1] ← delta_sum_xy[:, 1] + output[:, 2 * (i - 1) + 1]
        coor_future[:, 2 * i] ← coor_future[:, 2 * i] - delta_sum_xy[:, 0]
        coor_future[:, 2 * i + 1] ← coor_future[:, 2 * i + 1] - delta_sum_xy[:, 1]
    end
    loss ← loss_function(output, coor_future)
End

This approach aims to alleviate the impact of increasing prediction errors over time, potentially resulting in more favorable loss values during training.

IV. Training and Testing

During the training process, two common issues often arise. Firstly, the L2 loss curve may exhibit heavy redundancy, failing to converge to a sufficiently low value. The phenomenon is caused by selecting a batch size that is too small (less than 4 in this project), leading to excessive randomness and making it difficult for the model to converge.

Secondly, the training procedure may become trapped in local minima, hindering further progress. The phenomenon is caused by selecting a batch size that is too large (greater than 48 in this project), resulting in improved training speed and smoother loss curves. However, due to insufficient randomness, it becomes challenging to escape from capturing only the local minimum values.

As a result, the batch size is chosen to be 36 and works well. The optimizer here is the SGD optimizer and a changing learning rate is set as algorithm 11.

Algorithm 11 Algorithm for changing learning rate

Input:
cfg.epochs, cfg.final_lr (0.0001)
Output:
learning rate
Function `main()`**:**
    lf ←lambda x:
     $(1 - \frac{x}{\text{cfg.epochs}}) \times (1.0 - \text{cfg.final\_lr}) + \text{cfg.final\_lr}$
    scheduler ←
     torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda=lf)

Here is a table for test results:

methods		L2(m)			Collison(%)		
time		1s	2s	3s	1s	2s	3s
image and coor.	one lstm + dx + 1:1	0.53	1.18	2.1	0.28	1.87	5.1
	one lstm + dx + 12:96	0.32	0.60	1.00	0.52	0.65	0.99
	one lstm + or. + 12:96	1.38	6.6	15.38	0.6	3.69	5.43
	multi lstm + or. + 12:96	1.37	6.54	15.21	0.55	3.69	5.54
	multi lstm + dx + 12:96	0.31	0.62	1.06	0.33	0.52	1.01
pure coor.	one lstm + dx	0.25	0.56	1.01	0.34	0.49	0.83
	one lstm + or.	1.45	6.82	15.77	0.46	3.64	5.24
	multi lstm + or.	1.53	7	16.3	0.23	3.60	4.93
	multi lstm + dx	0.34	0.65	1.09	0.32	0.47	0.81
ST-P3		1.33	2.11	2.9	0.23	0.62	1.27

TABLE II: test results (dx means using modified L2; or. means using normal L2; 1:1 and 12:96 are the ratio between image features and coordinate features)

V. Conclusion

Based on the table, incorporating images as inputs only slightly reduces the L2 loss and fails to effectively decrease the collision rate, contradicting the initial assumption. As the proportion of image features increases during fusion with coordinate features, the model's performance deteriorates.

This suggests that ResNet50 cannot adequately extract the necessary image features, and the features it extracts may even degrade the model's performance.

After applying a custom L2 calculation method, both the loss value and collision rate significantly decrease, indicating the effectiveness of this approach. This outcome aligns with the initial hypothesis. Furthermore, cyclically predicting trajectory point coordinates does not effectively reduce the loss value and collision rate.

We found that incorporating images as inputs only marginally reduces the L2 loss and has no significant effect on decreasing the collision rate, which contradicts our initial hypothesis. Moreover, as the proportion of image features increases during fusion with coordinate features, the model's performance deteriorates. This suggests that ResNet50 may not effectively extract the required image features and the features it extracts may even worsen the model's performance.

However, after implementing a custom L2 calculation method, we observed a substantial decrease in both the loss value and collision rate, validating the effectiveness of this approach. This finding is consistent with our initial hypothesis. Additionally, cyclically predicting trajectory point coordinates did not lead to a notable reduction in the loss value or collision rate.

VI. Discussion

In theory, images should have a positive impact on reducing collision rates rather than increasing them, but experimental results do not align with this expectation. The speculated reason for this discrepancy is the ineffectiveness or poor performance of the network used to extract features from images. To address this issue in the future, it may be beneficial to utilize more complex networks for processing images, thereby effectively extracting their features and improving the performance of autonomous driving systems.

References

References

[1] Hu, Shengchao, et al. "St-p3: End-to-end vision-based autonomous driving via spatial-temporal feature learning." European Conference on Computer Vision. Cham: Springer Nature Switzerland, 2022.

[2] NuScenes Datset
$https://www.nuscenes.org/nuscenes?tutorial=nuscenes$

[3] View Geometry – Basics
$https://taotaoorange.wordpress.com/2011/05/09/view-geometry-basics/$.

[4] Hu, P., Huang, A., Dolan, J., Held, D., Ramanan, D.: Safe local motion planning with self-supervised freespace forecasting. In: CVPR (2021)

[5] Zeng, W., Luo, W., Suo, S., Sadat, A., Yang, B., Casas, S., Urtasun, R.: End-to-end interpretable neural motion planner. In: CVPR (2019)

[6] The Annotated ResNet-50

[7] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90.

[8] Website: structure of Resnet.

[9] LSTM Networks

[10] Deep Learning— Introduction to Long Short Term Memory
$https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/$

[11] Explaining L1 and L2 regularization in machine learning
$https://medium.com/@fernando.dijkinga/explaining-l1-and-l2-regularization-in-machine-learning-2356ee91c8e3$