２０２２年度　修士論文

Deep Learning-based Trajectory Prediction using Improved HiVT

指導教員　李　羲頡　教授

早稲田大学大学院情報生産システム研究科

情報生産システム工学専攻　システム制御研究

44211032-5　　SUN Honglin

Contents

Abstract

For safely driving in a dynamic environment, predicting the future trajectory of surrounding vehicles, pedestrians and other traffic participants is crucial for self-driving vehicles. Just as human drivers, by constantly observing the surrounding vehicles and pedestrians and guessing their intentions to adjust their driving. For trajectory prediction tasks, conventional methods such as physics-based models can predict future trajectories in a short term. But for long-term prediction, it is necessary to incorporate historical trajectories of traffic participants, environment, and interaction information (context information). Currently, the challenge is how to perform long-term multimodal (multi-trajectory) prediction based on this contextual information. Recently, deep learning-based trajectory prediction is emerging, showing surprising accuracy compared to conventional methods. However, the existing methods generally focus on higher accuracy and seldom consider computational consumption. For practical deployment, the trajectory prediction model needs to be able to predict online in real-time, otherwise, the prediction is meaningless. Thus, aiming at high-speed prediction, this paper proposes an improved HiVT (Hierarchical Vector Transformer) that significantly increases computational speed and reduces GPU memory cost while even improving accuracy compared to the original model.

Keywords: Trajectory Prediction, Motion Forecasting, Autonomous Driving, Transformer, Graph Neural Network, Deep Learning

Chapter 1
Introduction

1.1 Background

It has been nearly a century since the idea of autonomous vehicles was first depicted in movies in the early 1900s. in 1984, the first self-sufficient and truly autonomous vehicles were born at Carnegie Mellon University's NavLab. Then, since 2004, with the opening of the DARPA (Defense Advanced Research Projects Agency) Grand Challenge, autonomous driving began booming. Today, autonomous driving has become an essential component of modern vehicles.

In 2014, the SAE (Society of Automotive Engineers) classified driving automation into 6 levels [1], as shown in Figure 1.1. Level 2 and below are also known as assisted driving, where drivers still have to drive the vehicle. And level 3 still requires the driver to take over the control in the scenarios that the autonomous driving system cannot handle. Thus, Level 4 and above can be regarded as autonomous driving, which is what the academy and the industry are currently trying to overcome.

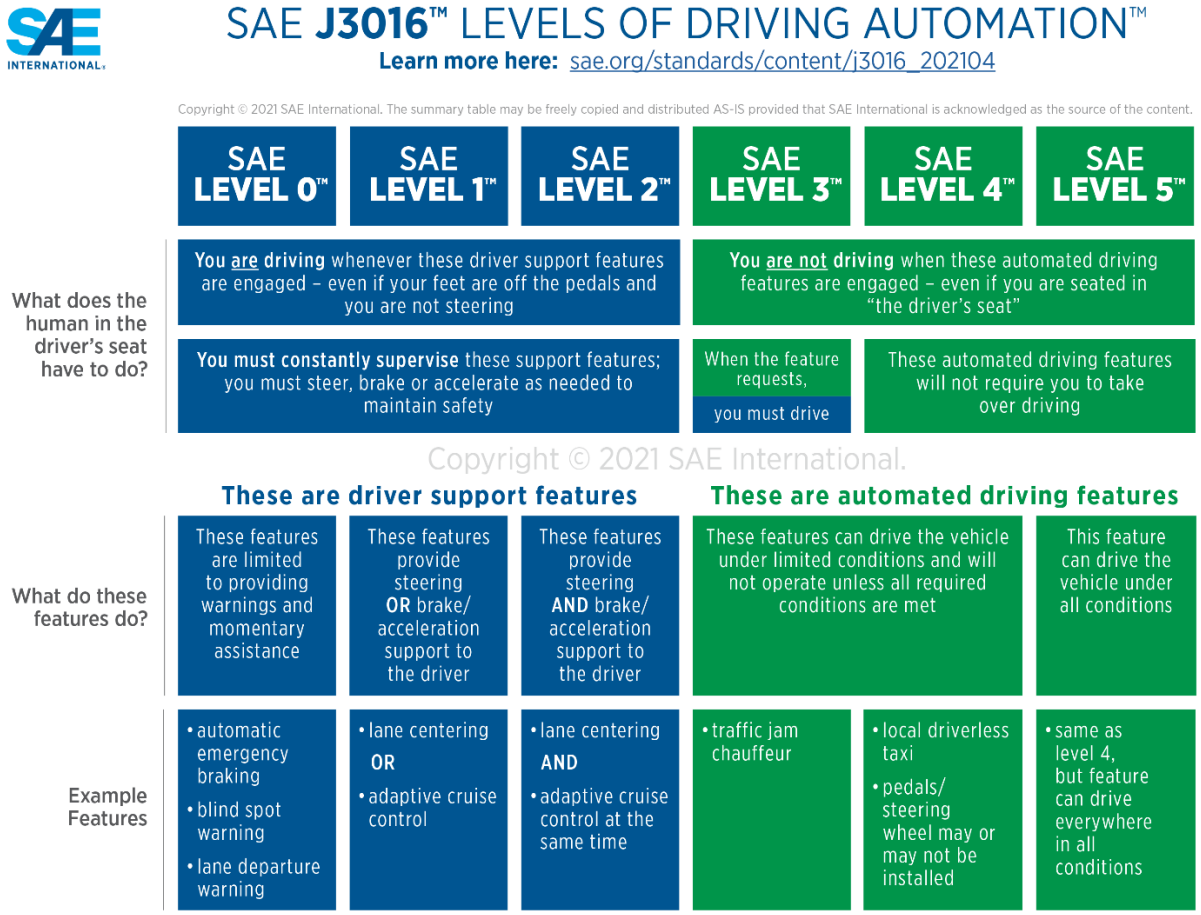


Fig. 1.1: Levels of driving automation

According to the reports of NHTSA (National Highway Traffic Safety Administration), 2,046,000 of the 2,189,000 traffic accidents they counted were caused by driver error, up to 94% [2]. In such background, autonomous driving is an effective solution to reduce accidents. Besides, it can also bring a better passenger experience, increase efficiency, save labor costs, etc.

The reason why high-level autonomous driving has not been widely applied so far is that, except for the safety risks caused by the technical limitations, there are still regulations, responsibility attribution, and social and ethical issues. Despite these challenges, driving automation is still an absolute trend.

As the most well-known company in autonomous driving, Google Waymo's fully autonomous driving vehicles were reported in 62 crashes in 2021 [3]. Also, as a pioneer in autonomous driving, Tesla vehicles running Autopilot (Level 2 autonomous driving) were reported in 273 crashes in 2021 [4]. It is obvious that, among the many issues, the most urgent one is safety.

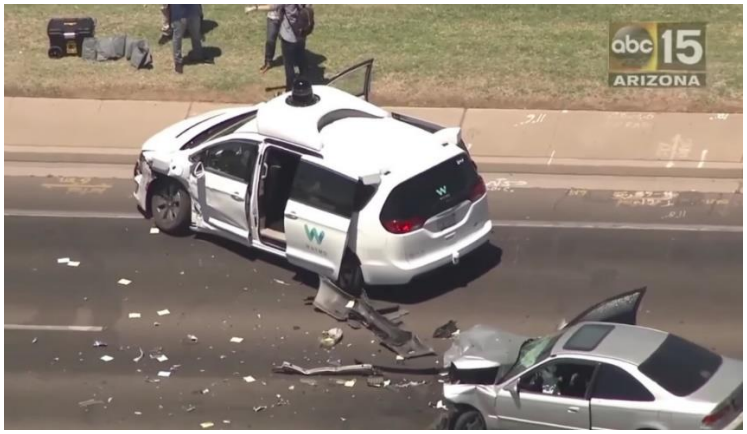


Fig. 1.2: Waymo car crash in Arizona

The autonomous driving framework can mainly be divided into modules including perception, localization, prediction, routing, behavior decision, motion planning, and control. Starting from the input, sensors continuously collect environmental information, send it to the perception module to analyze the environmental information, and finally, the motion planning module generates motion series and outputs control commands.

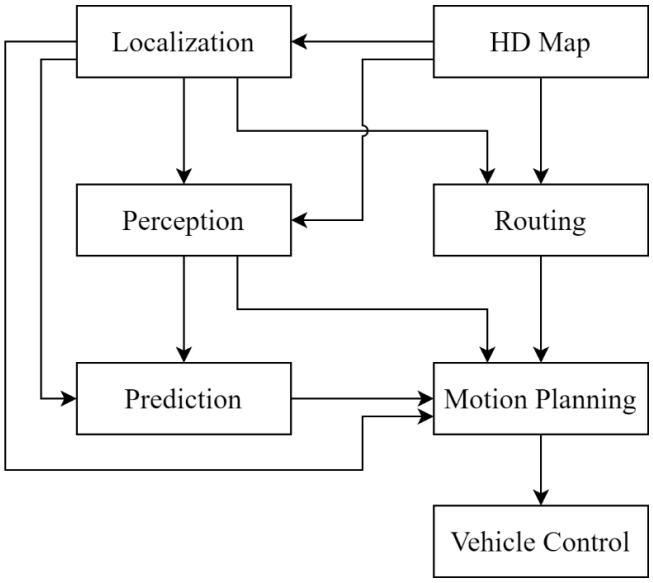


Fig. 1.3: Example of the autonomous driving framework

In particular, since motion planning is located at almost the end of the whole framework, its output directly affects driving safety, thus need to ensure that the process of getting the output is transparent and interpretable. Therefore, the motion planning module usually prefers rule-based methods, such as the Frenet trajectory generator [5], lattice planner [6], etc.

Whereas these rule-based methods have been developed for a long time, it is difficult to make a big breakthrough in the short term. While the recent emerging end-to-end learning-based planning [7] is hardly expected to be usable for the moment. Instead, before the planning module, the prediction was seldom focused on previously but could help the planning module to improve its safety. Also, for ADAS (Advanced Driver Assistance System), predicting the trajectory of other vehicles can help drivers avoid the risk of collisions. Nowadays, with the rise of deep learning, prediction algorithms are starting to take on a new dimension.

In the autonomous driving framework, trajectory prediction is generally located between the perception module and the planning module. Its input is the state information of the surrounding objects and environment by the perception module, integrating the semantic information provided by the HD (high definition) map. Outputs the possible future behaviors (cut in, go straight, etc.) or/and trajectories.

After prediction, the following module can generate motions for the vehicle, or alert the driver of possible risks, which is the meaning of trajectory prediction.

1.2 Issues of Previous research

Trajectory prediction also emerged very early and has developed for two decades so far. The methods of trajectory prediction can be roughly divided into two types, physics-based and learning-based. In the early stage, physics-based methods were mostly used, such as those using kinematic models [8]. This method aims at predicting the trajectory of vehicles, using the bicycle model, integrated with Kalman linear filter. Which considers the uncertainty of positioning (predicted positions) while keeping a low computational consumption. However, it can only predict one possible trajectory. In many cases, the behavior of the surrounding vehicles is highly uncertain, and it is difficult to predict a single trajectory to cover many possible behaviors. Also, the interaction and environment information cannot be modeled. Those limitations make the error large in the long-term prediction, which is a common problem for physics-based methods.

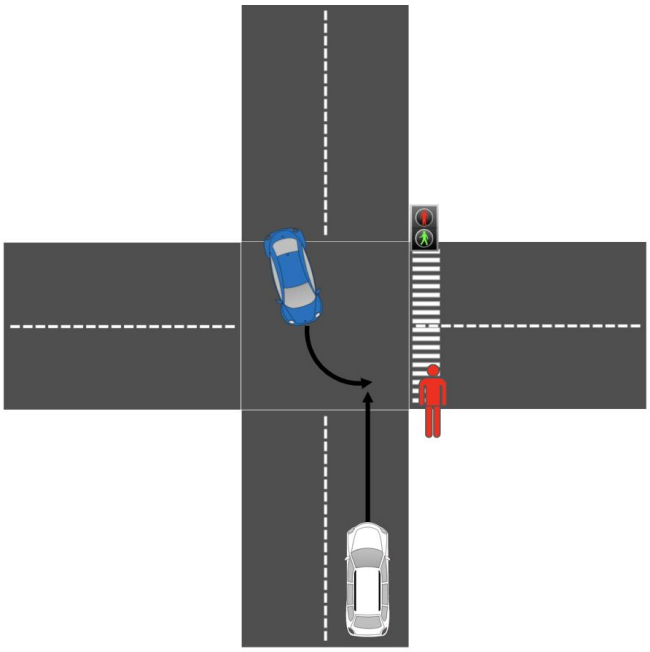


Fig. 1.4: Illustration of trajectory prediction

With the rise of deep learning and the explosion of CV (computer vision), deep learning-based trajectory prediction started booming in 2020. CoverNet [9] is a trajectory prediction network using CNN (Convolutional Neural Networks). In this network, the HD map is used. Compared with conventional physics-based methods, this method introduces rich environmental information and realizes multimodal prediction (predict multiple possible trajectories), ensuring accuracy even for long-term predictions. However, since the input to the network is images, due to the limitations of hardware performance, the size of input images to the CNN should not be too large, which means a compromise in the image resolution. Also, the semantic information that can be captured from images is very limited.

Another branch is using GNN (Graph Neural Networks) for prediction. GNN is widely used in fields such as personalized recommendation and searching. An important reason is that these data are difficult to be represented in images, but a kind of graph. Among them, LaneGCN [10] is one of the most famous models that use GNN for trajectory prediction. This model represents the HD map by vectors instead of rasterized images, solving the limitation of the input image size of CNN without information loss, leading LaneGCN to achieve state-of-the-art accuracy in the context of that time. However, the complex network structure results in a large number of parameters and heavy computation. It increases the computation time and memory consumption for the prediction. And this is a common problem of deep learning-based prediction. Especially for vehicles, it is difficult to equip high-performance computing devices that can cope with such computation due to cost and power consumption.

Meanwhile, methods using the attention mechanism began to be proposed. Transformer [11] has proven its outstanding performance in fields like NLP (natural language processing) and CV, thus it is a reasonable attempt to apply it to trajectory prediction. HiVT (Hierarchical Vector Transformer) [12] is a recent work based on GNN and Transformer. This model uses multiple

stacked transformer modules, and efficiently encodes the features of traffic participants, which achieved good accuracy with fewer parameters, greatly simplifying the network structure. But on the one hand, the simplification of the network and the reduction of the number of parameters make accuracy slightly lower. On the other hand, there are still inefficient operations in the network, which makes the prediction speed not high enough.

Generally, deep learning-based methods are believed to solve the issues of conventional physics-based methods. However, because of the brief history of development, most studies still focus on improving accuracy but seldom consider computational efficiency. Recently, some researchers noticed these issues and developed methods with a small number of parameters, but most of them are compromised in accuracy, while the computational speed is still not satisfactory.

1.3 Research Purpose

This research is based on the previously mentioned HiVT model since it achieves good accuracy with a relatively small number of parameters. In this paper, the HiVT model is modified in two stages to solve the common problem of the high hardware cost of learning-based methods.

In the first stage, the number of parameters is reduced by directly cutting out some parameters and layers in the network, and by reducing the network width. Then, by simplifying the network structure and modifying some complicated operations in the network, the number of parameters is further reduced, and the computational efficiency is increased. The modified network will significantly improve the computational speed and reduce the GPU memory cost, but at the same time, it will bring some accuracy degradation.

In the second stage, the network is finely complicated, including adding additional layers and computations where necessary, introducing more useful features, thus compensating for the accuracy loss without increasing the computational burden too much. Finally, adjust the training strategy to maximize the potential of the network. The modified network will ensure that the computational efficiency is still outperforming the original HiVT without sacrificing accuracy.

The rest of the paper is organized as follows: Chapter 2 will start with the introduction of some previous methods, followed by a detailed explanation of the HiVT model and the improvements made in this paper. Chapter 3 will first introduce the dataset used in this study, and in the simulation results section, the implementation details will be released, and the effectiveness of the proposed method will be demonstrated and analyzed through visualization and quantitative comparison as well as ablation experiments. Chapter 4 will conclude the whole paper.

Chapter 2
Deep Learning-based Trajectory Prediction

This chapter will describe the problem of trajectory prediction and give a general introduction to the previous conventional prediction methods referring to the classification in [13] and [14].

2.1 Problem Formulation

Generally speaking, the task of trajectory prediction is to predict future states based on the historical states of traffic participants. Given the historical states $\boldsymbol{s}_h$ for t_h seconds, the future state $\boldsymbol{s}_f$ for t_f seconds is predicted. Depending on the specific prediction method, the state $\boldsymbol{s}$ may contain coordinates, velocity, etc. In addition, learning-based methods may also consider the context information c, including the state of surrounding objects, lane information, semantic information of the map, etc. In general, it can be expressed as follows.

Consider that each traffic participant has historical information $\boldsymbol{x} = (\boldsymbol{s}_h, \boldsymbol{c}_h)$, where $\boldsymbol{s}_h = [s_1, s_2, \dots, s_{t_h}]$ and $\boldsymbol{c}_h = [c_1, c_2, \dots, c_{t_h}]$

Then, all the N traffic participants in the scenario compose a set $X = [\boldsymbol{x}_1, \boldsymbol{x}_2, \dots \boldsymbol{x}_N]$.

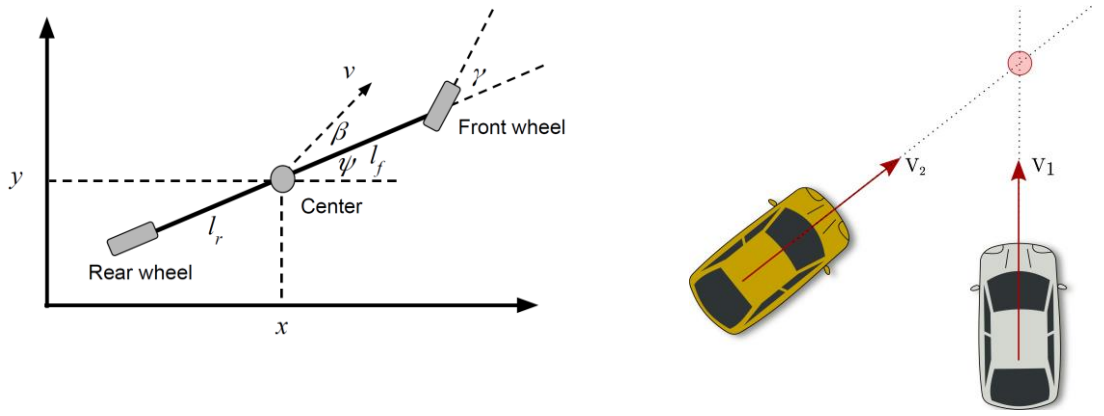
The prediction model $\mathcal{F}(\cdot)$ yields $Y = [\boldsymbol{y}_1, \boldsymbol{y}_2, \dots, \boldsymbol{y}_N]$, where $\boldsymbol{y} = \left[s_{t_h+1}, s_{t_h+2}, \dots s_{t_h+t_f}\right]$. Integrating the above expressions, get $Y = \mathcal{F}(X)$

2.2 Conventional Trajectory Prediction Methods

Conventional trajectory prediction methods can be classified into two categories: physics-based and classic machine learning-based.

2.2.1 Physics-based methods

It is intuitive to predict future positions based on physics assumptions using information such as the velocity and position of the object. Thus, in the early stage, trajectory prediction was usually performed with kinematic or dynamic models under the assumptions like constant velocity, constant acceleration, or constant steering speed. However, on the one hand, its prediction can only be unimodal for vehicles with a definite kinematic model, and cannot handle pedestrian motion or predict vehicle intentions. In addition, the kinematic assumptions can yield reliable results in the short term, but when the time is increased, the kinematic assumptions will no longer hold.

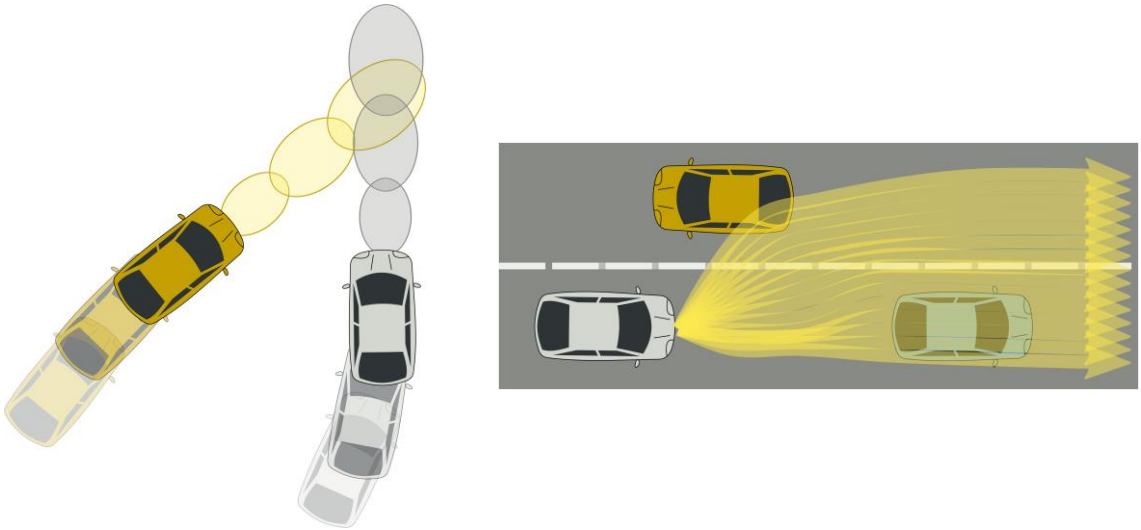


(a) Bicycle model (b) Prediction with constant velocity

Fig. 2.1: Trajectory prediction with the kinematic model

The subsequently emerged Kalman filtering methods model the uncertainty of vehicles as a Gaussian model. Compared with the physical model, noise is taken into account in the prediction, and thus the reliability of the prediction results is improved.

Another is the Monte Carlo method, which samples possible regions based on velocity and acceleration to generate predicted trajectories. Compared with Kalman filtering, Monte Carlo methods consider the kinematics of vehicles.



(a) Kalman filtering (b) Monte Carlo

Fig. 2.2: Trajectory prediction with Kalman filtering and Monte Carlo methods

In summary, these methods realize predictions with low computational cost, but the lack of understanding of traffic rules and maps, as well as the difficulty in modeling vehicle interactions, limit the accuracy to only be acceptable in short-term predictions.

2.2.2 Classic machine learning-based methods

Rather than completely relying on physical models, learning-based methods are data-driven and make predictions by exploiting features in the data, leading the prediction time to be longer and more stable. They are also referred to as maneuver-based methods described in [13].

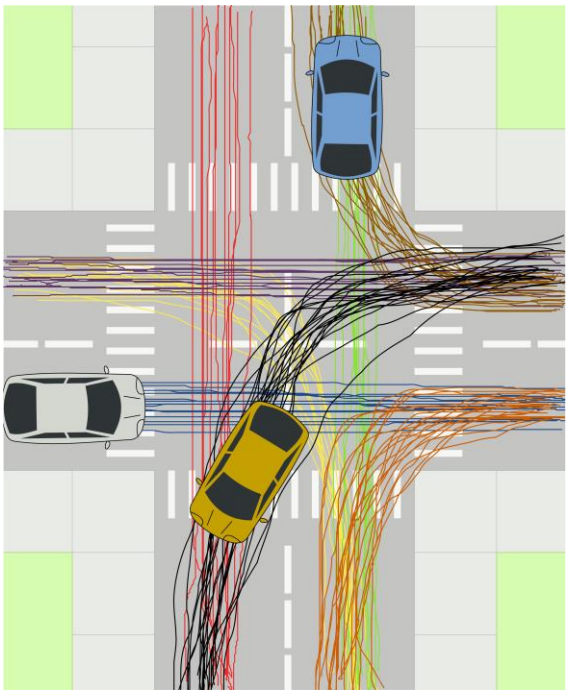
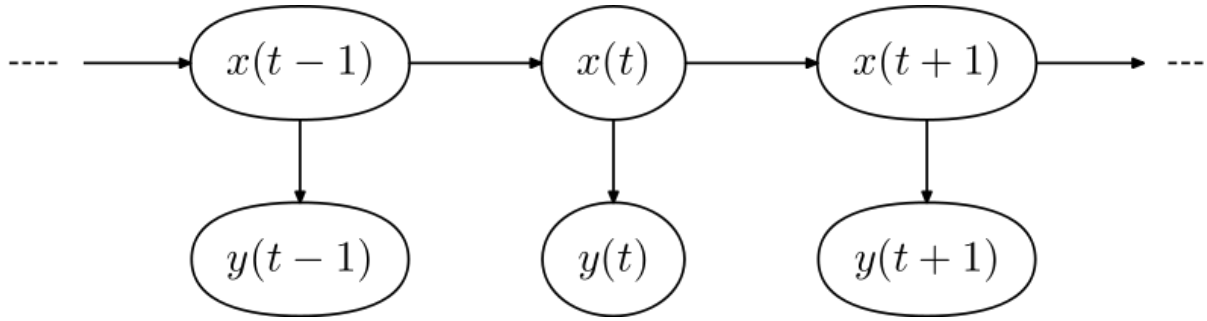


Fig. 2.3: Prototype trajectories (the same color indicates the same cluster)

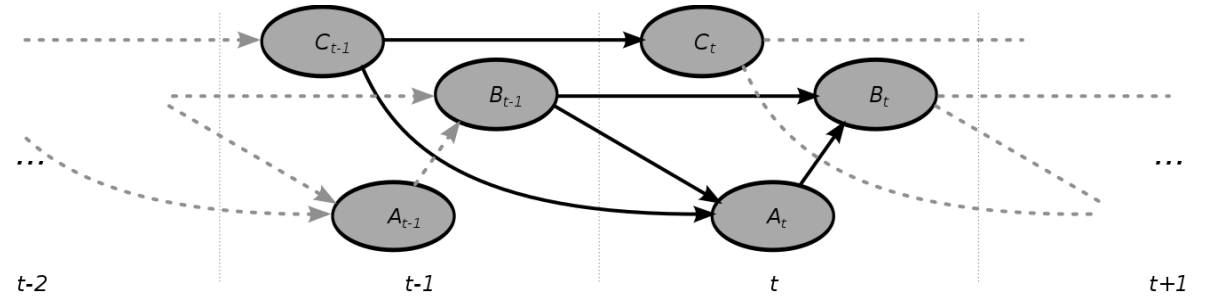
Gaussian process (GP) is a type of stochastic process in probability theory. When using GP for trajectory prediction, vehicle trajectories are grouped into a set of clusters, each of which corresponds to a motion pattern, and these trajectories are called prototype trajectories, as the samples of GP. When performing prediction, the model evaluates the similarity between the vehicles' historical trajectories and the prototype trajectories to predict possible future trajectories. However, the prototype trajectories can only be used for a specific scenario during training, resulting in its poor generalization ability.

Hidden Markov model (HMM) is a method that uses Markov chains. Compared to the Markov model, not all states can be observed in HMM, the unobservable hidden states need to be predicted based on the observed states and Markov assumptions. When applied to trajectory prediction, the observable states generally refer to the states of objects surrounding the vehicle, and the hidden states generally refer to the intentions of surrounding objects output by the HMM. Although HMM can perform multimodal intention prediction, it still cannot take interactions between traffic participants into account.

To address this shortcoming, methods based on dynamic Bayesian networks (DBN) were developed. DBN can be seen as a generalized HMM. unlike the undirected graph of HMM, DBN is a directed graph model, i.e., it can consider the causality between states. When expanded over time, HMM can be represented as a transition from one hidden state to another, with each hidden state corresponding to an observable state. In contrast, a hidden state in DBN can be jointly determined by multiple known states, i.e., there can be several traffic participants in the state space. Based on this feature, the prediction model considering the interaction among traffic participants can be designed.



(a) Hidden Markov model



(b) Dynamic Bayesian network

Fig 2.4: Comparison of HMM and DBN

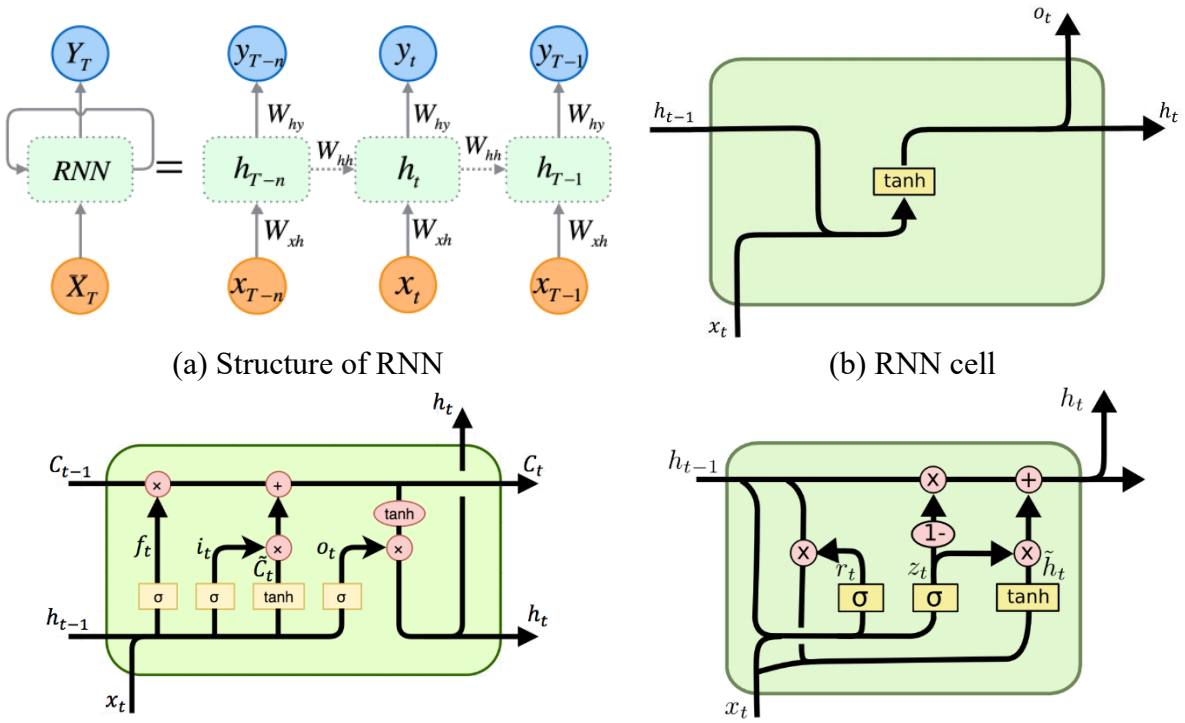
Compared to physics-based methods, these learning-based methods consider more factors in prediction, resulting in better accuracy. It also paves the stage for the deep learning-based methods that will come later.

2.3 Deep Learning-based Trajectory Prediction Methods

As conventional methods only have good results for simple scenarios and short-term predictions, deep learning-based trajectory prediction is starting to be studied more and more with the success of deep learning in various fields. By the network structure designing, more factors, such as the map and semantic information, can be considered to achieve higher accuracy in long-term prediction. This research also adopts deep learning-based trajectory prediction, hence this section will focus on the development of deep learning-based prediction methods.

2.3.1 RNN and CNN-based methods

The RNN (Recurrent Neural Networks) series of methods, such as LSTM (Long Short-term Memory), and GRU (Gated Recurrent Unit) mining temporal information in data through the recurrent structure, have achieved good results in sequence prediction, such as speech recognition, word processing, etc.



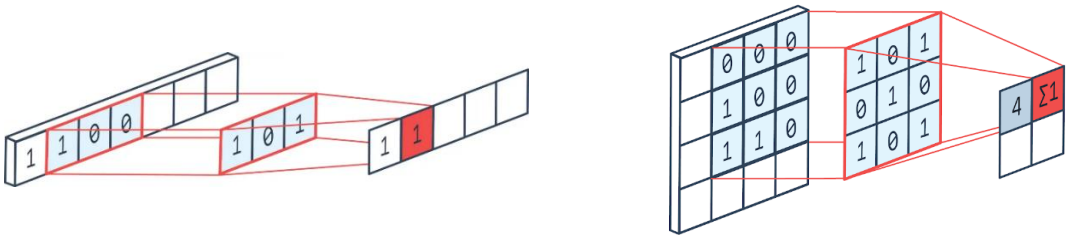
(a) Structure of RNN (b) RNN cell

(c) LSTM cell (d) GRU cell

Fig 2.5: Structure of RNN series network

Since trajectory is naturally a kind of time-series data, deep learning-based prediction with RNN was first started to be studied. For example, [15] is an early work based on LSTM. It uses encoder-decoder architecture and results in a more than 80% reduction in MAE (mean average error) in 2-second prediction compared with Kalman filtering.

Subsequently, CNN, popular in the field of computer vision, was also tried to be applied to trajectory prediction.



(a) 1D-CNN (b) 2D-CNN

Fig 2.6: CNN for 1D and 2D input

Representing the information contained in the scenario as a bird's-eye-view is an intuitive attempt, and this is the underlying logic used by CNN for trajectory prediction. For example, in CoverNet, information such as historical trajectories of traffic participants, drivable areas, and crosswalks are represented on images with different colors and encoded by CNN. Then, HOME [16] encodes trajectories and the context information separately, using 2D CNN for the context information and 1D CNN for the trajectories.

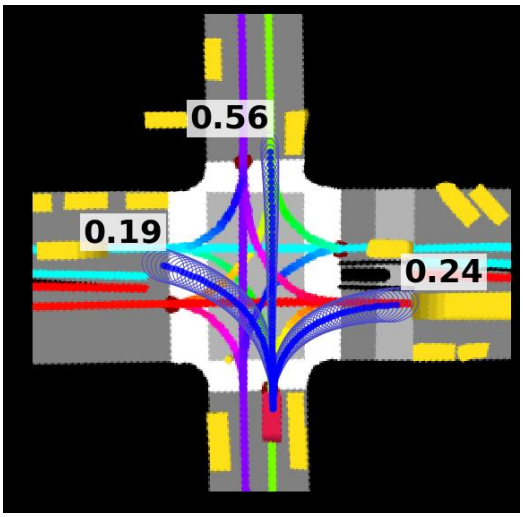


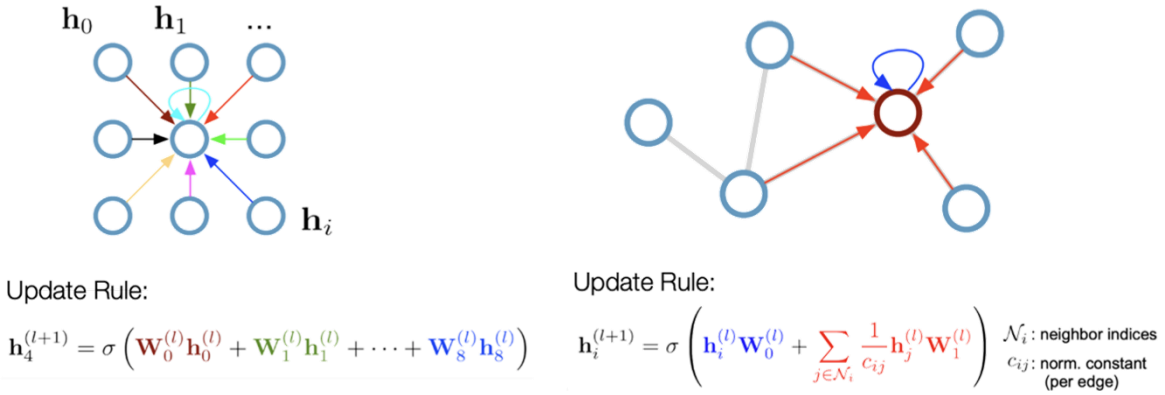
Fig. 2.7: Image input to the network with prediction output

Other than only considering accuracy, [17] proposed Deep Kinematic Model (DKM), which uses kinematic models in learning-based trajectory prediction. It uses RNN to encode and decode trajectories and CNN to encode context information. The output is acceleration and steering angle at each time step, and the trajectories are calculated using the bicycle model. Which ensures the trajectory is kinematic compliant.

These methods are good attempts at early-stage deep learning-based trajectory prediction, having a better performance in long-term prediction than conventional methods.

2.3.2 GNN-based methods

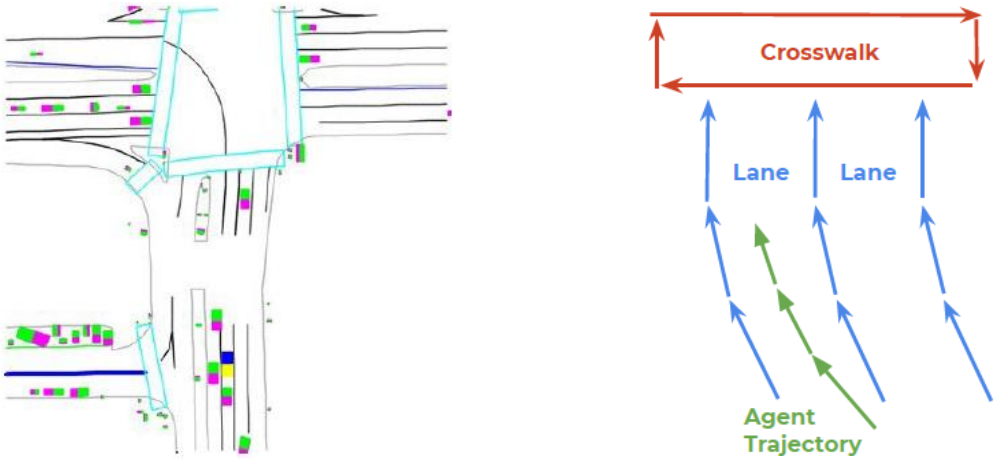
One of the keys to the success of CNN is that it can handle structured data well, in which, images can be seen as a kind of rasterized structure. However, information such as lane coordinates, traffic participant positions, and semantic information like traffic lights and lane directions are essentially a kind of unstructured data, and representing these data in images will cause information loss. For example, when the positions of traffic participants are marked on the images, their accuracy will be limited by the resolution of the images. As a result, GNN-based prediction networks were soon proposed.



(a) Convolutional neural network (b) Graph neural network

Fig 2.8: Difference between CNN and GNN

The most different point from RNN and CNN-based methods, GNN-based methods represent the map as a graph. VectorNet [18] is the pioneering work on using vectorized maps and GNN for trajectory prediction, aiming to solve the shortage of CNN methods in input resolution. It represents the information of lanes, trajectories, and crosswalks in polylines, which are composed of several subgraphs, and uses GNN to encode and process the interactions between the subgraphs. By using GCN (Graph Convolutional Networks), the network can capture the topological relationship in the vectorized map and efficiently model the interactions between traffic participants and the environment. This information is easily represented by graphs, and capturing the relationships in the graph is exactly what GCN is good at.



(a) Rasterized representation (b) Vectorized representation

Fig. 2.9: Map with different representations

LaneGCN, which appeared almost at the same time, has done more work on capturing the topological relations of lanes, and the LaneConv operator is proposed to encode the different adjacency relations between lane nodes.

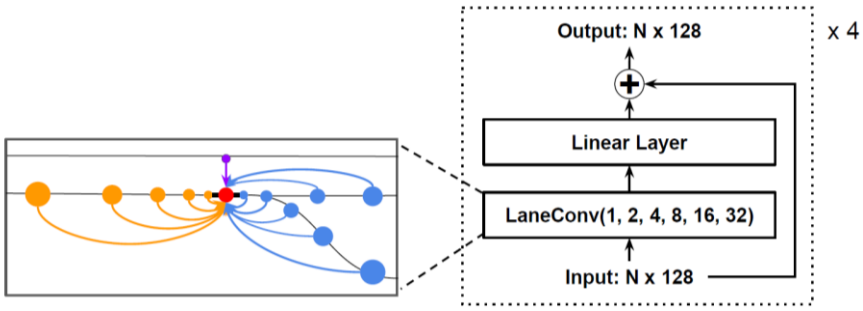


Fig. 2.9: LaneGCN block with LaneConv operator

In the LaneConv operator, long-range information along the lane direction is captured by using the multi-scale dilated convolutions [19], and the trajectory information of the traffic participants is encoded by FPN [20]. It first divides the interactions in the scenario into four types: Actor-to-Lane, Lane-to-Lane, Lane-to-Actor, and Actor-to-Actor, and proposes the fusion net to handle these interactions. It was so influential in the following methods that both the winners in 2021 [21] and 2022 [22] of the Argoverse Motion Forecasting Challenge used LaneGCN-inspired variants.

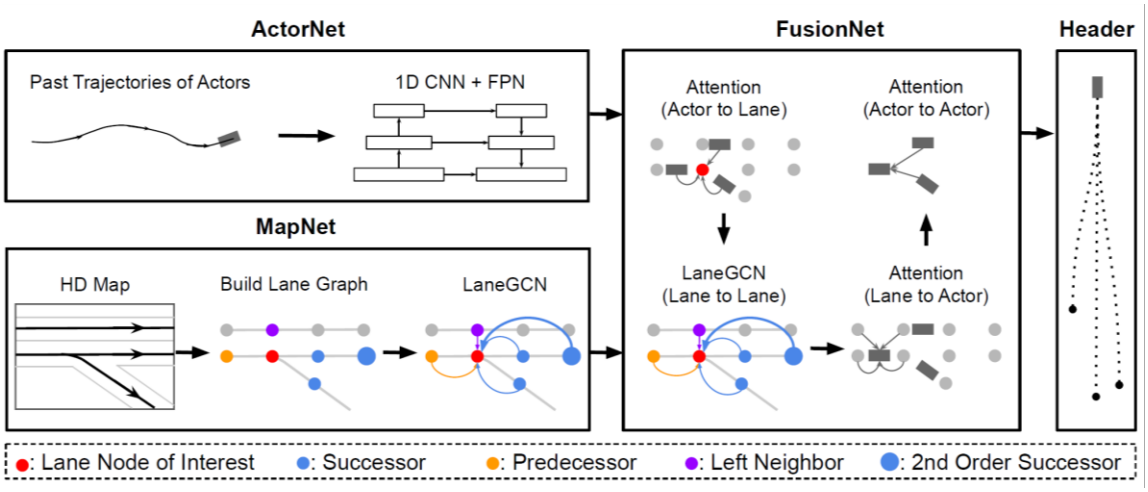


Fig. 2.10: Overall structure of LaneGCN

Another model, TNT [23], worked on the decoding method. Inspired by anchor-based methods in image recognition (e.g., YOLOv4 [24]), the anchor is sampled on the lane centerline. The confidence scores of anchors are output by the classification network, and the regression network is to output the offset of the final position from the anchor to get the target coordinates. Then targets with the highest confidence scores are selected for completing the full trajectories. The authors point out that this method is to decompose the uncertainty of the future states into the intent (target) uncertainty and the motion (how to arrive at the target) uncertainty. Then, DenseTNT [25], their improved version that comes later, achieved first place in the Waymo Open Dataset Motion Prediction Challenge in 2021.

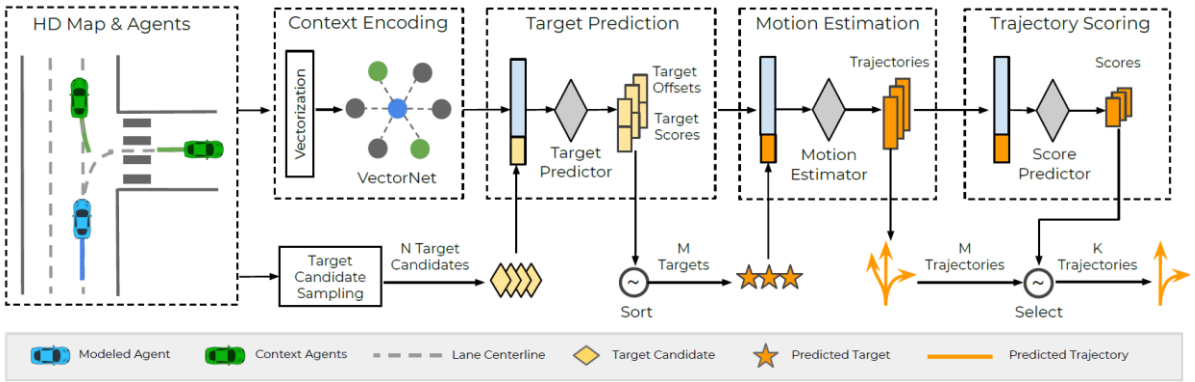


Fig. 2.11: Overall structure of TNT

With continuous exploration, methods using vectorized maps and GNN gradually replace rasterized maps and CNN and become mainstream.

In 2017, Google proposed the Transformer model based on the Attention mechanism, which completely replaced CNN and RNN as the most powerful feature extractor in the field of NLP (Natural Language Processing). In 2020, ViT (Vision Transformer) [26] was proposed to bring the Transformer into the field of CV, proving that it is equally good in areas other than NLP. In this background, the combination of GNN and transformer is natural.

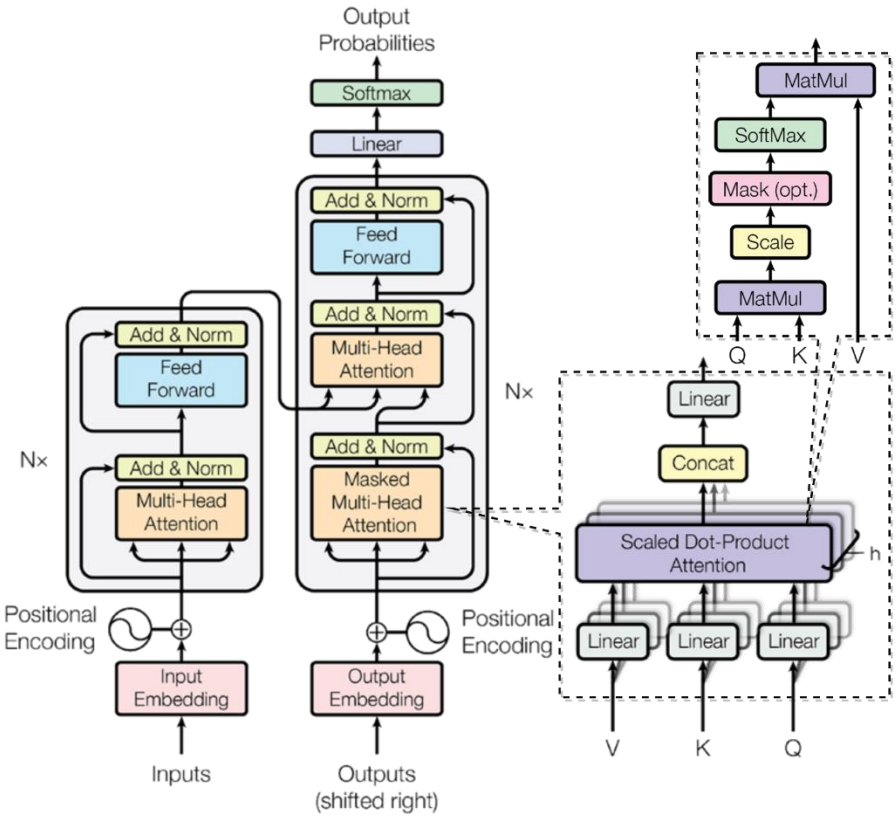


Fig. 2.12: Structure of Transformer and scaled dot-production attention

Among the many methods that use the Transformer's scaled dot-product attention, HiVT (Hierarchical Vector Transformer) is a relatively new study proposed in 2022. The detailed implementation of the method will be highlighted here, and the research in this paper will be based on the model.

The HiVT model first crops the scenario into several circular regions (named local regions) centered on each traffic participant (agent), which can be regarded as a kind of spatial attention, and the cropping operation reduces the computational complexity. Then, the authors designed an encoder-decoder architecture, where the encoder contains a local encoder and a global interaction module. The local encoder encodes the trajectories, lanes, and other information contained in each local region to obtain local features. The global interaction module puts each local feature back into the whole scenario to encode the global interaction to compensate for the range limitation on cropping the local regions, thus yielding the global features. Finally, the local features and global features are fed into the decoder together to obtain the predicted trajectory. In addition, HiVT utilizes a Gaussian mixture model (GMM) as the trajectory output, outputting both the coordinates of each position and the corresponding uncertainty. The output is modeled as a Laplace distribution (L1 distribution), and the probability density function of the Laplace distribution is regressed by using negative log-likelihood (NLL) loss.

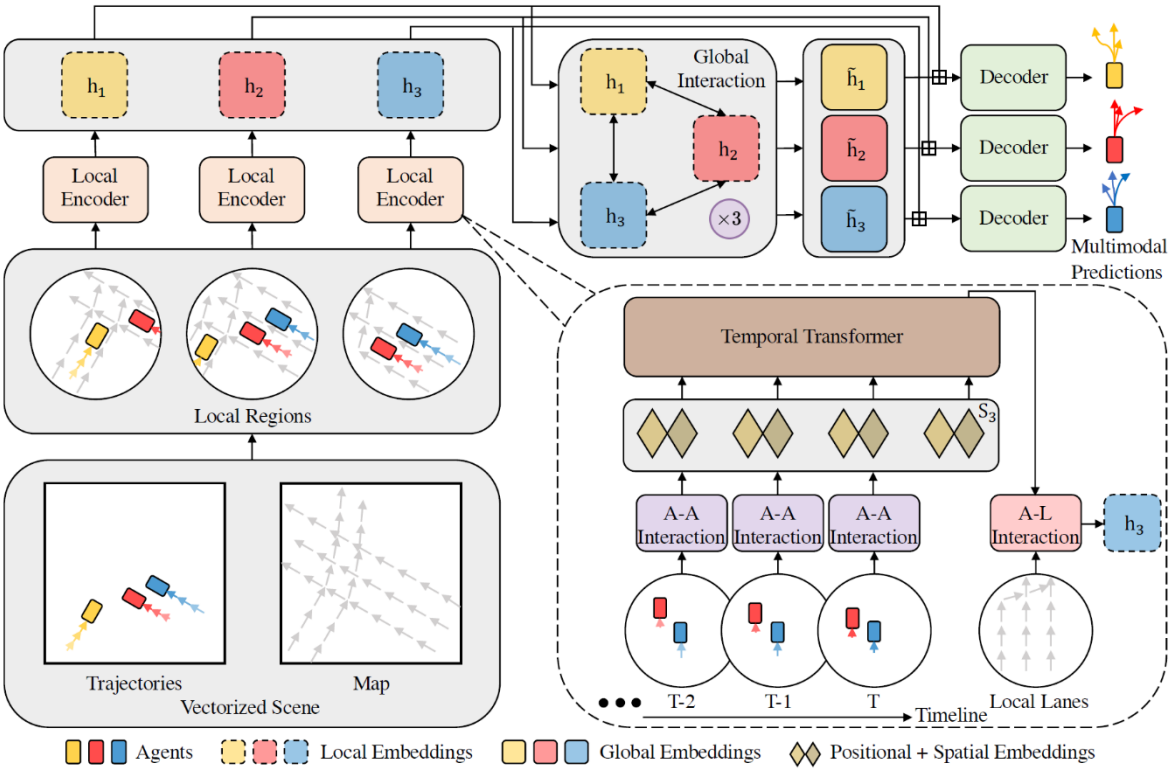


Fig. 2.13: Overall structure of HiVT (⊞ denotes concatenation)

Concretely, in the local encoder, HiVT uses a structure with 3 stacked Transformers, as in Figure 2.13. First, encode the interactions between agents in each local region at each time step using the agent-to-agent (A-A) module with multi-head cross-attention. Then, the information in the time domain is extracted in the temporal Transformer module by using self-attention, which compresses the time dimension to 1. Finally, the lane features are fused into the agent features in the agent-to-lane (A-L) module by using cross-attention again.

The global interaction module consists of a 3-layer stacked global interactor layer, which is also implemented by multi-head cross-attention. The relationship of coordinate transformation between each local region is encoded and input to the global interactor layer together with all local features. The obtained global features are projected to multiple dimensions through a linear layer for multimodal prediction, each dimension corresponds to one modality, i.e., trajectory.

Finally, in the decoder, the local features are expanded to the same dimension and concatenated with the global features. The trajectories, uncertainties, and confidence outputs are performed by MLP (multi-layer perceptron).

This study is conducted based on HiVT, on the one hand, it is believed that Transformer and GNN are currently the most promising and successful solutions for trajectory prediction. On the other hand, as shown in Table 2.1, HiVT outperforms many high computational complexity models with high hardware requirements, using a smaller number of parameters and achieving state-of-the-art accuracy.

TABLE 2.1: Comparison of learning-based methods (LaneGCN as the baseline of accuracy)

Method	Accuracy	#Param.	Device (GPU)
LaneGCN [10]	100.00%	3,701K	GTX TITAN X
Scene Transformer [27]	110.71%	15,296K	Tesla V100
GOHOME [28]	94.05%	400K	RTX 2080 Ti
HiVT-64	99.77%	653K	RTX 2080 Ti

2.4 Proposed Improved HiVT

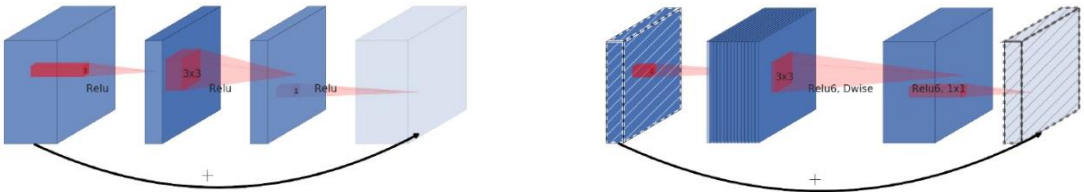
Although HiVT has a great advantage over other deep learning-based methods in terms of the number of parameters and computational complexity. It is still difficult to meet the requirements for real-time prediction when applied in practice on low-performance devices. For example, the AI chip that is currently being used in production vehicle XPeng P7, the Nvidia Jetson AGX Xavier, has only about 10% computing power for FP32 (32-bit floating-point number) of the device used in the HiVT paper (Nvidia RTX 2080 Ti).

2.4.1 Computational efficiency improvement

In the original paper of HiVT, the authors provided two widths of models, HiVT-128 and HiVT-64. In tests, the inference speed of HiVT-64 is only 19.11 fps*. Thus, the priority is to reduce the amount of computation to increase the inference speed of the model.

(1) Cut down parameters and layers.

Originally, the feed-forward network inside the Transformer applied an inverted bottleneck structure, where in the middle, the number of feature channels is increased to four times than the input and output, which is a common setting in Transformer-based models. Here reducing the width by half, getting an inverted bottleneck with twice the width. This step reduces the number of parameters by 148K.



(a) Bottleneck (b) Inverted bottleneck

Fig. 2.14: An example of bottleneck structure

Then, reduce the number of global interactor layers in the global interaction module from 3 to 1, which further cut down 117K parameters.

Finally, remove the biases in some layers. In the HiVT model, the bias is kept in all linear layers, where the bias performs a translation over the linear projection by the weights. In practice, the linear layer is mostly followed by normalization, which will pull the input forcefully back to a normal distribution with a mean of 0 and variance of 1, eliminating the translation by bias.

* Tested with Nvidia GTX 1070 Max-Q

In the mathematical expression, a linear layer with bias has $\boldsymbol{y} = W\boldsymbol{x} + \boldsymbol{b}$, where W is the weight and $\boldsymbol{b}$ is the bias. The normalization has the formula $\boldsymbol{y} = \frac{\boldsymbol{x}-\mathrm{E}(\boldsymbol{x})}{\sqrt{\mathrm{var}(\boldsymbol{x})+\epsilon}}\boldsymbol{\gamma} + \boldsymbol{\beta}$, where $E(\cdot)$ calculates the mean value, $\mathrm{var}(\cdot)$ calculates the standard deviation, ϵ is a small quantity for avoiding division by 0, γ and β are learnable parameters.

First, the bias does not affect the standard deviation. Then, when computing the numerator, substitute the input containing the bias, get

$$\begin{aligned}\mathcal{F}(W\boldsymbol{x} + \boldsymbol{b}) &= W\boldsymbol{x} + \boldsymbol{b} - \mathrm{E}(W\boldsymbol{x} + \boldsymbol{b}) \\ &= W\boldsymbol{x} + \boldsymbol{b} - \mathrm{E}(W\boldsymbol{x}) - \boldsymbol{b} \\ &= W\boldsymbol{x} - \mathrm{E}(W\boldsymbol{x}) \\ &= \mathcal{F}(W\boldsymbol{x})\end{aligned}$$

where the bias is also canceled out in the numerator. So, this proves the rationality of removing the bias in the linear layer before the normalization layer.

Based on a similar logic, the bias in the linear layer for the projection of Q, K, V matrices in the Transformer can also be removed. After getting the Q, K, V matrices, the scaled dot-product attention block in the Transformer will calculate as

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^{\top}}{\sqrt{d_k}}\right)V$$

Where,

$$\mathrm{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Take q_i, k_j, from Q, K matrices, if enable bias, then

$$\begin{aligned}q_i \cdot k_j &= \left(w_Q x_i + b_Q\right) \cdot \left(w_K x_j + b_K\right) \\ &= w_K x_j\left(w_Q x_i + b_Q\right) + b_k\left(w_Q x_i + b_Q\right)\end{aligned}$$

Plugging them into the $\mathrm{softmax}(\cdot)$, get

$$\begin{aligned}\mathrm{softmax}\left(q_i \cdot k_j\right) &= \frac{\exp\left(q_i \cdot k_j\right)}{\sum_j \exp\left(q_i \cdot k_j\right)} \\ &= \frac{\exp\left(w_K x_j\left(w_Q x_i + b_Q\right) + b_K\left(w_Q x_i + b_Q\right)\right)}{\sum_j \exp\left(w_K x_j\left(w_Q x_i + b_Q\right) + b_K\left(w_Q x_i + b_Q\right)\right)} \\ &= \frac{\exp\left(w_K x_j\left(w_Q x_i + b_Q\right)\right)\exp\left(b_K\left(w_Q x_i + b_Q\right)\right)}{\sum_j \exp\left(w_K x_j\left(w_Q x_i + b_Q\right)\right)\exp\left(b_K\left(w_Q x_i + b_Q\right)\right)} \\ &= \frac{\exp\left(b_K\left(w_Q x_i + b_Q\right)\right)\exp\left(w_K x_j\left(w_Q x_i + b_Q\right)\right)}{\exp\left(b_K\left(w_Q x_i + b_Q\right)\right)\sum_j \exp\left(w_K x_j\left(w_Q x_i + b_Q\right)\right)} \\ &= \frac{\exp\left(w_K x_j\left(w_Q x_i + b_Q\right)\right)}{\sum_j \exp\left(w_K x_j\left(w_Q x_i + b_Q\right)\right)}\end{aligned}$$

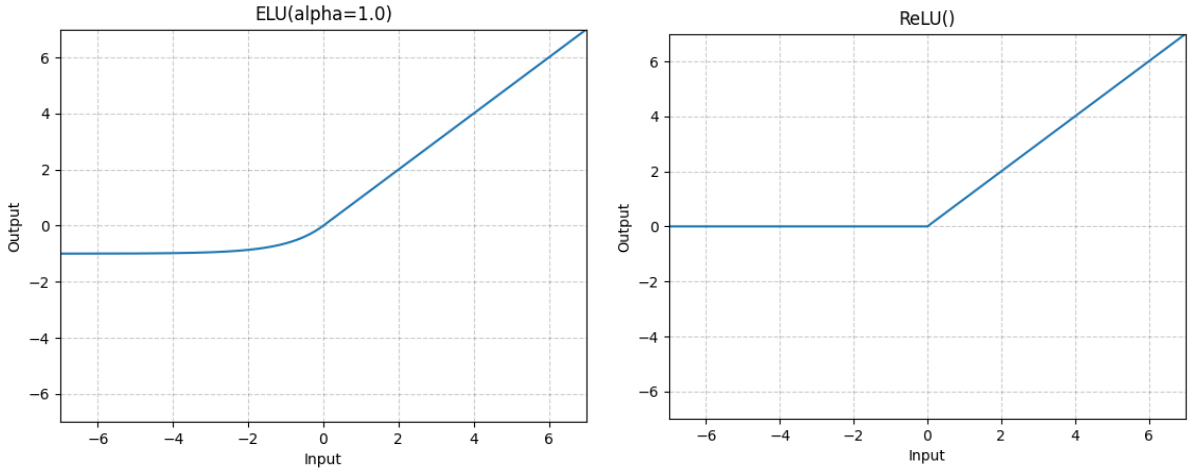
It can be seen that, after $\mathrm{softmax}(\cdot)$, the bias b_K of K is eliminated. In recent studies, many transformer-based models have removed all the biases of Q, K, V. In this paper, it is also found that removing these biases has almost no effect on the performance after experiments.

(2) Modify or remove some operations.

The modifications to the operations are mainly in the decoder. When outputting the uncertainty of each position, the original model uses $\mathrm{ELU}(\cdot)+1$ to transform the output from the network into an uncertainty in the range $(0,+\infty)$. However, $\mathrm{ELU}(\cdot)$ contains the exponential operation, which will increase the computational load. Also, the value domain of $(0,+\infty)$ makes the calculation result of NLL loss would become negative. Here, $\mathrm{ELU}(\cdot)+1$ is replaced by $\mathrm{ReLU}(\cdot)+1$. The equations and images of $\mathrm{ELU}(\cdot)$ and $\mathrm{ReLU}(\cdot)$ are as follows.

$$\mathrm{ELU}(x)=\begin{cases}x, & x>0\\ \alpha(\exp(x)-1), & x<0\end{cases}$$

$$\mathrm{ReLU}(x)=\begin{cases}x, & x>0\\ 0, & x<0\end{cases}$$



(a) $\mathrm{ELU}(\cdot)$ (b) $\mathrm{ReLU}(\cdot)$

Fig. 2.15: Comparison of $\mathrm{ELU}(\cdot)$ and $\mathrm{ReLU}(\cdot)$

Then, change the fusion method of the local and global features in the decoder that was originally concatenated to addition. As concatenation will double the width of channels in the feature dimension, increasing the input size of the following linear layers and increasing the number of parameters. While using addition keeps the width of channels constant, which reduced 3.7K parameters. Moreover, making the regression and classification networks share the same feature matrix output from the feature aggregation block for better utilization.

(3) Implement a new local encoder.

Finally, the structure of the local encoder is modified by changing the original A-A interaction performed at each time step to be performed only once after the A-L interaction. The trajectory data of the agent at the center of each local region is extracted directly by stacked linear layers, named agent encoder.

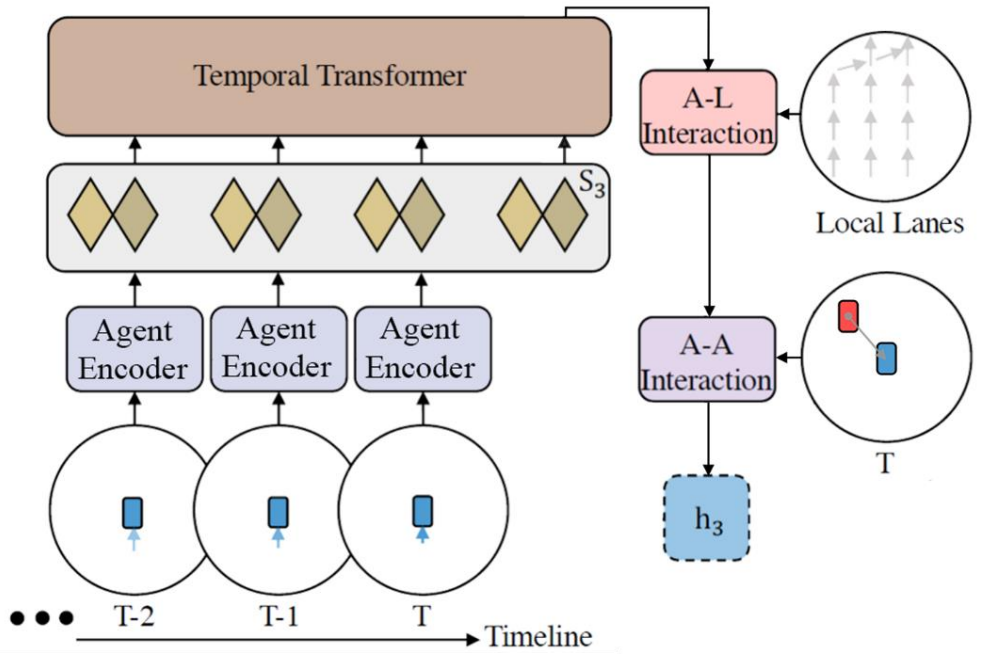


Fig. 2.16: Proposed new local encoder

The motivation for this modification is to consider that not all A-A interactions are noteworthy. For example, as shown in Figure 2.17, when the surrounding agents gradually move away from the center agent or even disappear from the local region, at which point it can be assumed that there is almost no interaction between them. Similarly, if the surrounding agent passes quickly on the opposite lane of the center agent, it may only appear for a few frames in the local region, which is also an example of almost no interaction. So, it can be believed that in this case, much of the computation in the A-A module is meaningless. In contrast, by placing the A-A module after the A-L module, each agent holds richer features and can consider the semantic information of the map when processing A-A interactions, which may compensate for the potential accuracy loss.

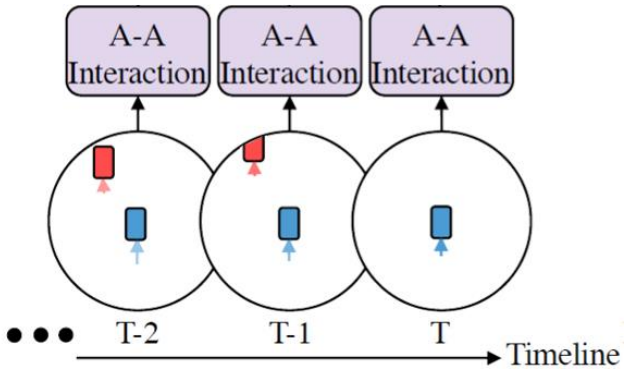


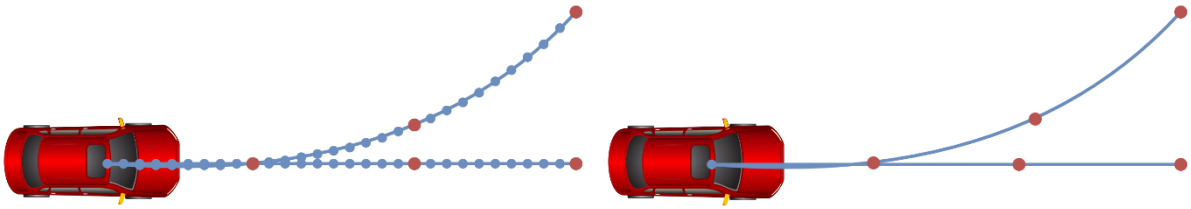
Fig. 2.17: Issues of A-A interaction module

2.4.2 Prediction accuracy improvement

(1) Refine the network structure.

Even for deep learning-based trajectory prediction, the kinematic compliance of the output trajectories should be considered. Directly using the network to output dense trajectory points

is difficult to ensure kinematic compliance, and the large output size will cause the network harder to converge. Inspired by the statement in TNT that "intentions in short time intervals can be considered as deterministic", the output of the decoder is changed from a dense output with 0.1-second intervals to a sparse output with 1-second intervals. The dense trajectory was subsequently interpolated by using cubic B-spline interpolation.



(a) Dense output (b) Sparse output with interpolation

Fig. 2.18: Different methods of trajectory generation

In the same process that sparse output followed by trajectory completion, this research adopts a 1-second interval output with interpolation rather than output the third-second position and completes the trajectory by the network as adopted in TNT. The rationality is that the 3-second horizon is difficult to ensure that the intention is deterministic for drivers or pedestrians, etc. In this case, completing trajectories with the network can be considered that the network still needs to learn the potential change of intentions of traffic participants during these 3 seconds. By contrast, interpolation of sparse locations with 1-second intervals is considered to completely leave the prediction of intentions to the network, and the motions to realize these intentions are derived using mathematical methods. The cubic B-spline was chosen because it is also frequently used in trajectory planning.

Besides, as the network goes deeper, the problem of network degradation causes deeper networks to have lower training accuracy on the contrary. ResNet [29], published in 2016, introduced the concept of identity mapping and proposed the residual network by stacking residual blocks to solve the network degradation problem. Now, the residual connection has become one of the most popular techniques to improve network performance. Here introduce residual connection in the linear layers in the decoder to construct a residual linear block as shown in Figure 2.19. The residual linear block is equivalent to the original two linear blocks* connected with a residual connection. It improves network accuracy without increasing the number of parameters, but only an extra addition.

* A linear block consists of a linear layer, a normalization layer, and an activation function.

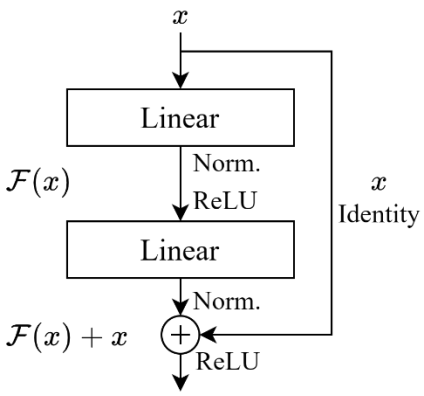


Fig. 2.19: Residual linear block

Then, carefully add more extra layers in some necessary places. In the decoder, since the aggregation of local and global features is previously changed from concatenation to addition, and the regression, classification network, and uncertainty estimation block share the same feature matrix, the feature aggregation block that generates this feature matrix is required to have a higher feature extraction ability. Here increase the number of layers of feature aggregation by one and increase the number of layers of the regression network to the same as the classification network.

Using the same idea in the encoder, add additional layers to the lane encoding block in the A-L interaction module and let the number of layers be the same as for the agent, considering that the lane features should be equally important as the trajectory. Overall, there are not too many layers added.

Finally, modify the usage of the edge attribute in the global interaction module. In the graph structure, an edge connects two nodes. In the context of this paper, a node is a local region, and an edge attribute (e) is a relative position relationship between two local regions, as shown in Figure 2.20. Here $\boldsymbol{e}_{ij} = [\boldsymbol{p}_{ij}, \theta_{ij}]$

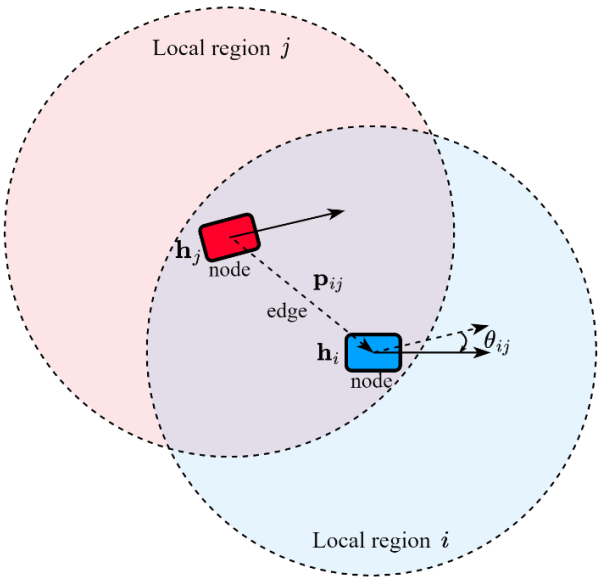


Fig. 2.20: Illustration of edge attributes

In the original global interaction module, the edge attributes are converted into feature embedding with MLP and fed into Transformer. Then, the edge embedding would be

$$\boldsymbol{e}_{ij} = \phi_{\mathrm{MLP}}\left(\left[R_{ij}\boldsymbol{p}_{ij}, \sin\left(\theta_{ij}\right), \cos\left(\theta_{ij}\right)\right]\right)$$

Where R_{ij} is the rotation matrix with an angle θ_{ij}.

In this case, the Q, K, and V in the calculation of the attention score can be obtained by

$$Q = W_{Q_{\mathrm{node}}}\boldsymbol{h}_i$$
$$K = W_{K_{\mathrm{node}}}\boldsymbol{h}_j + W_{K_{\mathrm{edge}}}\boldsymbol{e}_{ij}$$
$$V = W_{V_{\mathrm{node}}}\boldsymbol{h}_j + W_{V_{\mathrm{edge}}}\boldsymbol{e}_{ij}$$

Where $\boldsymbol{h}$ and $\boldsymbol{e}$ denote the feature matrix of the local region and the edge embedding between two regions.

The approach in this research is to fuse the edge attributes into the feature matrix of the local region, i.e., into the node, thus removing the terms of the edge attributes in the calculation of the attention score. The reason is that the original approach of adding K and V of nodes and edges directly is not so theoretically sound while fusing edge attributes into nodes retains more node information. After modification, $\boldsymbol{h}_j$ is calculated as

$$\boldsymbol{e}_{ij} = \left[R_{ij}\boldsymbol{p}_{ij}, \sin\left(\theta_{ij}\right), \cos\left(\theta_{ij}\right)\right]$$
$$\boldsymbol{h}_j = \phi_{\mathrm{MLP}}\left(\left[\boldsymbol{h}_j, \boldsymbol{e}_{ij}\right]\right)$$

Then, the calculation of Q, K, V becomes

$$Q = W_{Q_{\mathrm{node}}}\boldsymbol{h}_i$$
$$K = W_{K_{\mathrm{node}}}\boldsymbol{h}_j$$
$$V = W_{V_{\mathrm{node}}}\boldsymbol{h}_j$$

(2) Introduce more useful features.

In the original HiVT, only intersection, traffic signal, and turning directions are considered as the semantic information, but no further lane information. Also, the lane vectors of all directions in the local region are equally fused into the local feature matrix of the agent. One reason for the success of LaneGCN can be believed that it effectively captures lane node adjacencies, but implementing its LaneConv operator requires heavy computations. Hence, aiming at realizing the representation of lane adjacency information with minimum cost, this research exploits the number of the left and right lanes of lane segments from the dataset as the feature of the adjacency relationship. For instance, assuming a lane at the intersection that the number of adjacent lanes at the left and right sides is 1, then the network can easily catch the relationship of vehicles in such lanes with trajectories going straight through the intersection. An example of this adjacency relationship is shown in Figure 2.21.

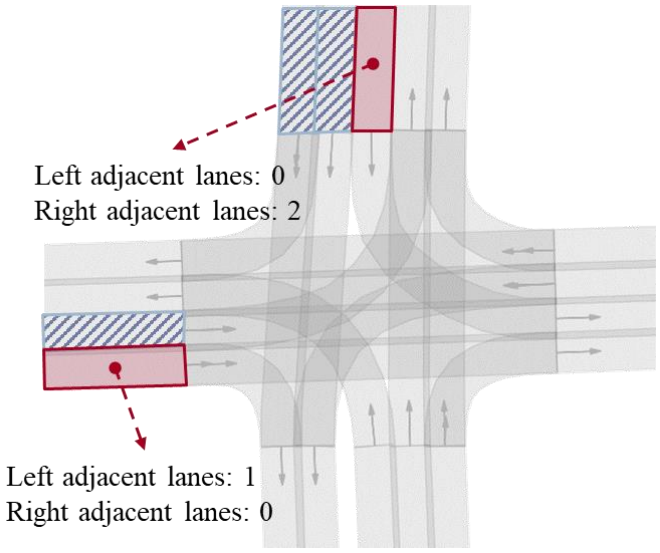


Fig. 2.21: Example usage of lane adjacency relationship

Empirically, there are rarely more than 5 lanes in the same direction, at which point the number of lanes on both the left and right side of the middle lane is 2. When the number of lanes increases, the middle lane will generally still be a straight-forward lane. Accordingly, the number of left and right adjacent lanes is limited here to a maximum of two. Specifically, the embedding layers are used to encode the left and right adjacency information separately, which is a discrete encoding method. Assuming the inputs can be 0, 1, or 2, the embedding layer contains three learnable vectors, one of which is selected to be the output according to the specific input. Comparatively, the use of linear layer coding requires converting the inputs to one-hot vectors to ensure that the inputs have the same distance in the feature space. The matrix multiplication is time-consuming compared to the indexing operation in the embedding layer.

Next, introduce learnable time weight to the input of the temporal Transformer. The temporal Transformer in HiVT refers to the design in ViT, using an additional `[class]` token to aggregate information, as shown in Figure 2.22. For image recognition, this approach allows the Transformer to extract information equally for each image patch in the spatial domain. However, in the temporal domain, it can be roughly assumed that newer information is more valuable, i.e., it should have a higher weight. For implementation, directly multiply the output of the agent encoder by the learnable weights, which are initialized with 1. The experiments show that after training for epochs, the first time step has a high weight, and from the second time step, the weights gradually increase from the lowest, with the last step having the highest weight. This is aligned with the initial hypothesis and proves the effectiveness of this idea.

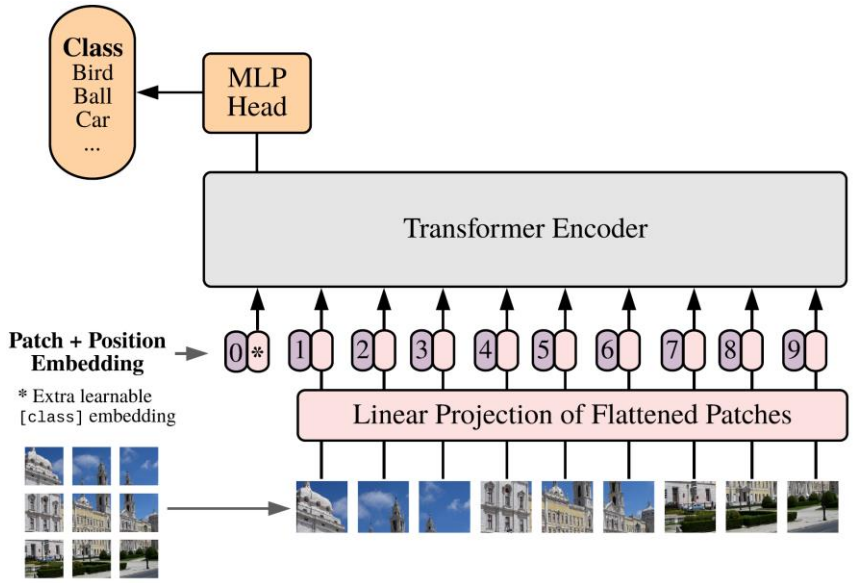


Fig. 2.22: Vision Transformer (* indicates extra `[class]` token)

Finally, integrate all the modifications. The overall structure of the network is shown in Figure 2.23.

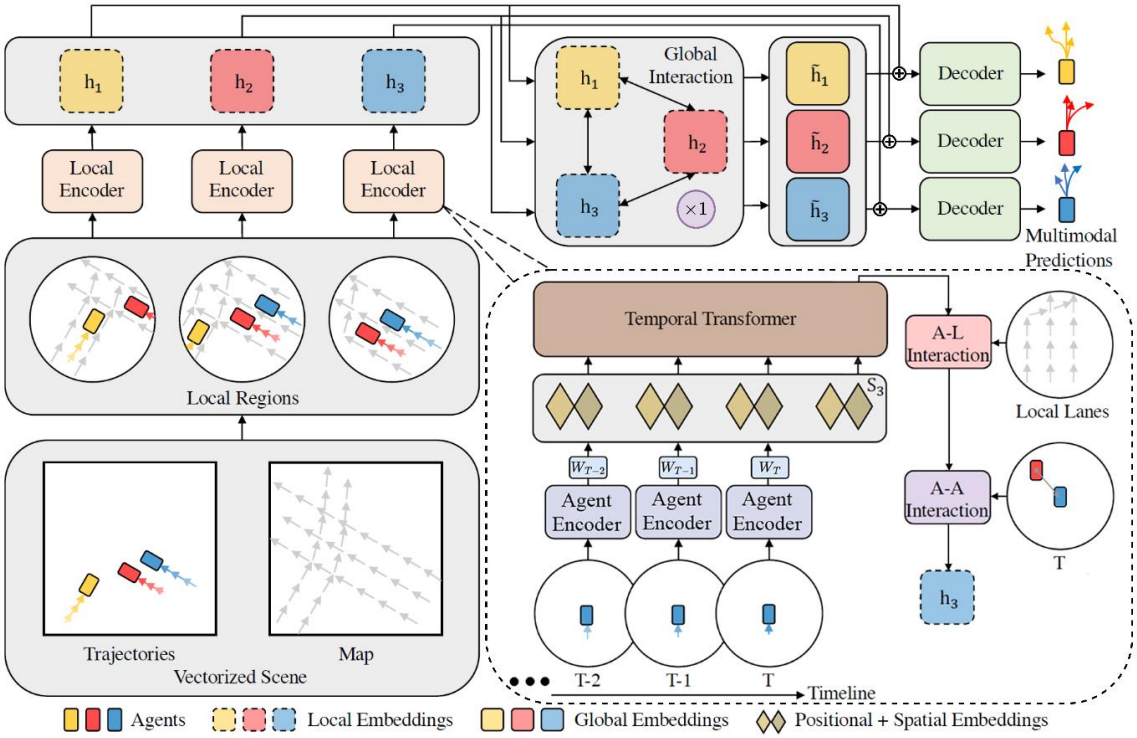
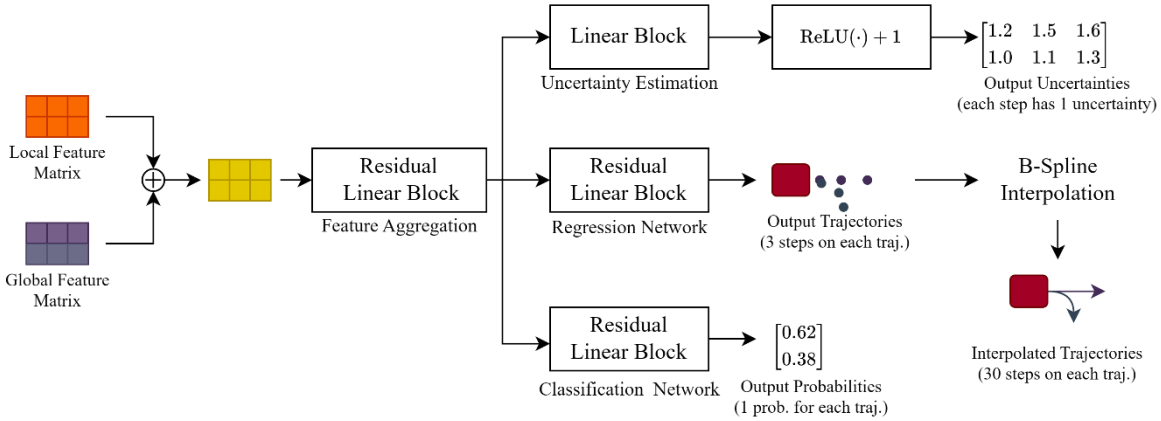


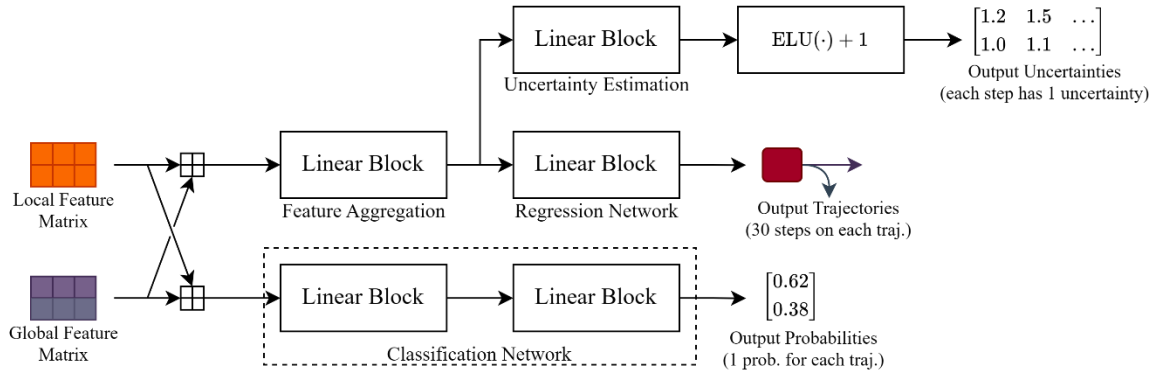
Fig. 2.23: Overall structure of the modified HiVT

The structure comparison inside the decoder is shown in Figure 2.24*.

* For clearer view, a linear layer with bias before the final output is omitted.



(a) Modified decoder (⊕ denotes addition)



(b) Original decoder (⊞ denotes concatenation)

Fig. 2.24 Structure inside the decoder

(3) Improve the training strategy.

The last is to achieve performance improvements by changing the training strategy while keeping the network completely unchanged, which is a common technique in deep learning. Using different parameter initialization methods, optimizers, and even random seeds can lead to different training results.

Here, the classification loss is first changed from cross-entropy loss to KL (Kullback-Leibler) divergence loss. In information theory, information entropy represents the uncertainty of information. The role of information is to reduce this uncertainty. For machine learning, the information entropy is defined as

$$H(X) = -\sum_{x \in X} p(x) \cdot \log\big(p(x)\big)$$

Where $p(x)$ is a probability distribution.

Given two distributions $p(x)$ and $q(x)$, the KL divergence loss is to measure the similarity of these distributions with the equation

$$D_{\mathrm{KL}}(X) = \sum_{x \in X} p(x) \cdot \log\left(\frac{p(x)}{q(x)}\right)$$

Suppose $p(x)$ is the true distribution to be learned and $q(x)$ is the output of the classification

network, changing the form slightly, get

$$\begin{aligned} D_{\mathrm{KL}}(X) &= \sum_{x\in X} p(x)\cdot\log\left(\frac{p(x)}{q(x)}\right) \\ &= \sum_{x\in X} p(x)\cdot\log\big(p(x)\big) - \sum_{x\in X} p(x)\cdot\log\big(q(x)\big) \\ &= -H(X) - \sum_{x\in X} p(x)\cdot\log\big(q(x)\big) \end{aligned}$$

It can be seen that it contains a term of information entropy of the true distribution. While the cross-entropy loss ignores this term $-H(X)$, which has

$$H_{\mathrm{CE}}(X) = -\sum_{x\in X} p(x)\cdot\log\big(q(x)\big)$$

This is based on an assumption that the true distribution is constant. However, for trajectory prediction, since the network needs to predict multiple possible trajectories while the ground truth trajectory has only one, the true distribution needs to be derived indirectly from the output positions of the regression network, and this distribution may be changed by the regression network. Specifically, it is calculated by a max entropy model, using the average displacement error between the predicted results and the ground truth trajectory.

$$p(\boldsymbol{s}) = \frac{\exp\big(-\mathcal{D}(\boldsymbol{s},\hat{\boldsymbol{s}})\big)}{\sum_i \exp\big(-\mathcal{D}(\boldsymbol{s}_i,\hat{\boldsymbol{s}})\big)}$$

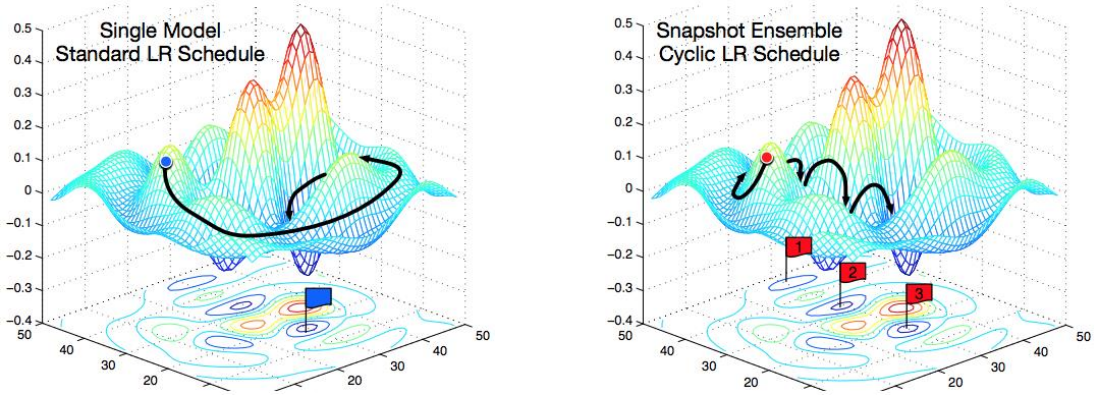
Where,

$$\mathcal{D}(\boldsymbol{s}_i,\hat{\boldsymbol{s}}) = \frac{1}{T}\sum_{i}^{T}\|s_i - \hat{s}_i\|_2$$

Then, when using KL divergence loss as classification loss, on the one hand, the true distribution is considered in the loss calculation, while on the other hand, this term reduces the volatility of the loss when the true distribution tends to be stable.

Then, for the modification of the learning rate function, the warmup mechanism is applied before training, which is a technique commonly used for Transformer training. In general, due to the attention mechanism, Transformer is more difficult to be trained than other models. Using a large learning rate at the beginning will make the update of the weights unstable and the model hard to converge [30]. So, in the early stage, the mechanism of warmup is used, making the learning rate increase gradually from smaller values.

After rising to the max learning rate, adopt cosine annealing with warm restarts [31] as the learning rate scheduler. This learning rate function, based on cosine annealing, periodically returns to the max learning rate every time it decreases to the minimum, which is called the warm restart. Compared to the original cosine annealing scheduler, the utilization of warm restarts enables the model easily to jump out of the current local optimum, as Figure 2.25 shows.



(a) Without warm restarts (b) With warm restarts

Fig. 2.25: Difference between with and without warm restarts

The modification of the learning rate function makes the training time longer, but it does improve the accuracy of the model without extra computation in inference, so it is worthwhile. Combining learning rate warmup and cosine annealing, the learning rate function is shown in Figure 2.26.

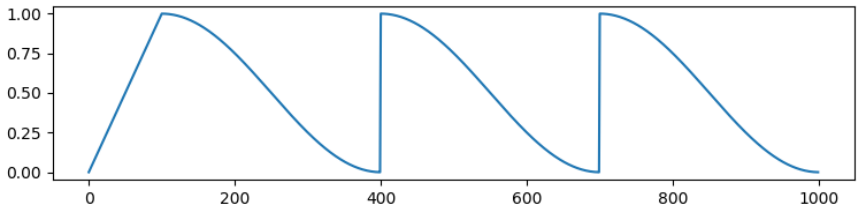


Fig. 2.26: Combine learning rate warmup and cosine annealing with warm restarts

Chapter 3
Simulations

This chapter will introduce the dataset used in this research, the implementation details, and the simulation results.

3.1 Argoverse Motion Forecasting Dataset

This research uses an open-source dataset, Argoverse Motion Forecasting Dataset [32], provided by Argo AI. The data was collected in Miami and Pittsburgh, with 324,557 scenarios in the dataset. 5 seconds of data for each scenario with a 10 Hz sampling rate. The trajectory of 0~2 seconds is used for prediction, and the task is to predict the future positions of agents in the next 3-second future horizon.

The dataset was divided into training, validation, and test sets with 205942, 39472, and 78143 scenarios, respectively. The test set is used for evaluating the metrics for the online leaderboard, with only two seconds of the historical trajectories publicly available. Therefore, in addition to calculating the metrics on the validation set locally, the results for the test set will also be submitted to the leaderboard for comparison.

In each scenario, besides the position of agents, the available information in the dataset includes the coordinates of the centerline of lane segments, and the corresponding predecessor, successor, left neighbor, and right neighbor lane segments for each lane segment. The centerline coordinates of each lane segment are represented in the order of their direction and contain information on whether the lane is at an intersection, whether it contains a traffic signal, and the turning direction of the lane segment.

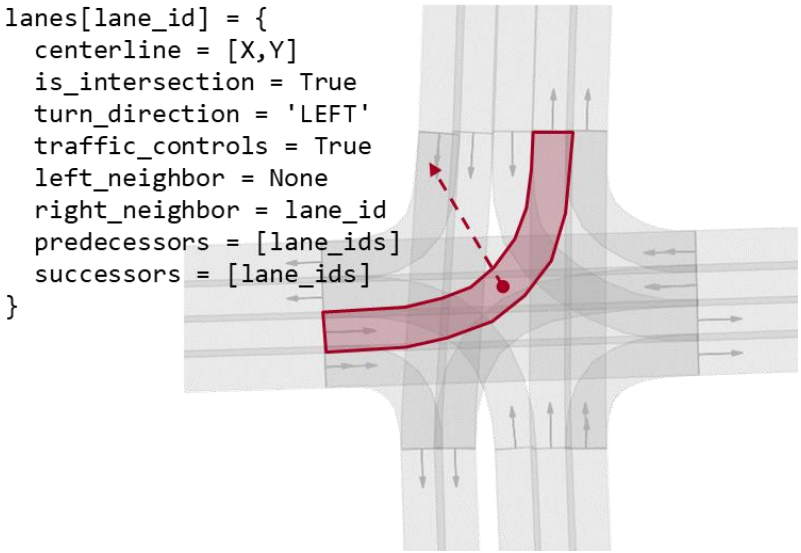


Fig. 3.1: Visualization of the map data

3.2 Metrics

The commonly used metrics to evaluate the prediction results include minADE (Minimum Average Displacement Error), minFDE (Minimum Final Displacement Error), MR (Miss Rate), and brier-minFDE (Brier Minimum Final Displacement Error) [33]. When only one trajectory

is output, the error of this trajectory is the minimum error. When multiple trajectories are output, the error of each trajectory is calculated and the prediction with the smallest error is selected. In the Argoverse Motion Forecasting Challenge, the official evaluation metrics have adopted minFDE, MR, and brier-minFDE. So, the comparison of these metrics will be shown in the simulation results. All these metrics evaluate errors, so all of them lower is better.

MR is the ratio of scenarios where the final position of the predicted trajectory differs from the ground truth position by more than two meters. Other metrics are calculated as follows.

$$\text{minADE} = \min\left(\frac{1}{T}\sum_{t=1}^{T}\|\hat{\boldsymbol{y}}_t - \boldsymbol{y}_t\|_2\right)$$

$$\text{minFDE} = \min(\|\hat{\boldsymbol{y}}_T - \boldsymbol{y}_T\|_2)$$

$$\text{brier-minFDE} = \min(\|\hat{\boldsymbol{y}}_T - \boldsymbol{y}_T\|_2 + (1 - \boldsymbol{p})^2)$$

Here, $\hat{\boldsymbol{y}}_t$ is the predicted position at time t, $\boldsymbol{y}_t$ is the corresponding ground truth position. T represents the final time step of the prediction. $\boldsymbol{p}$ is the probability of each trajectory. Hence, brier-minFDE is a metric that evaluates both regression and classification performance.

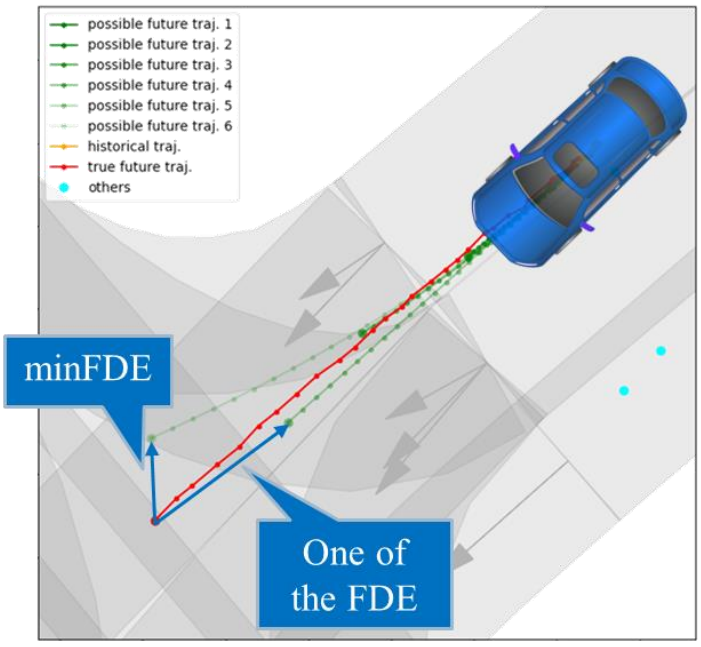


Fig. 3.2: Illustration of the FDE and minFDE for the prediction results

3.3 Implementation Details

The CPU used for training is Intel i7-8750H and the GPU is Nvidia GTX 1070 Max-Q, where, the GPU used in this research is about 4 times the performance of Nvidia Jetson AGX Xavier and 42% of the GPU used in the HiVT paper. The model is built using the deep learning framework PyTorch 1.12 with CUDA 11.6, running on Ubuntu 20.04 LTS.

The hidden size of the network is set to 64. Then adopt the AdamW [34] as the optimizer and cosine annealing with warm restarts as the learning rate scheduler. The first 4 epochs are for the

warmup stage, at which the learning rate increases linearly from 1×10^{-4} to 5×10^{-4}. Then switch to cosine annealing with warm restarts, set the first restart period to 1 epoch, and double the period after each restart. The training ends after 7 periods, and the last period contains 64 epochs. Therefore, the total number of epochs including warmup is 131. The final learning rate function is shown in Figure 3.3.

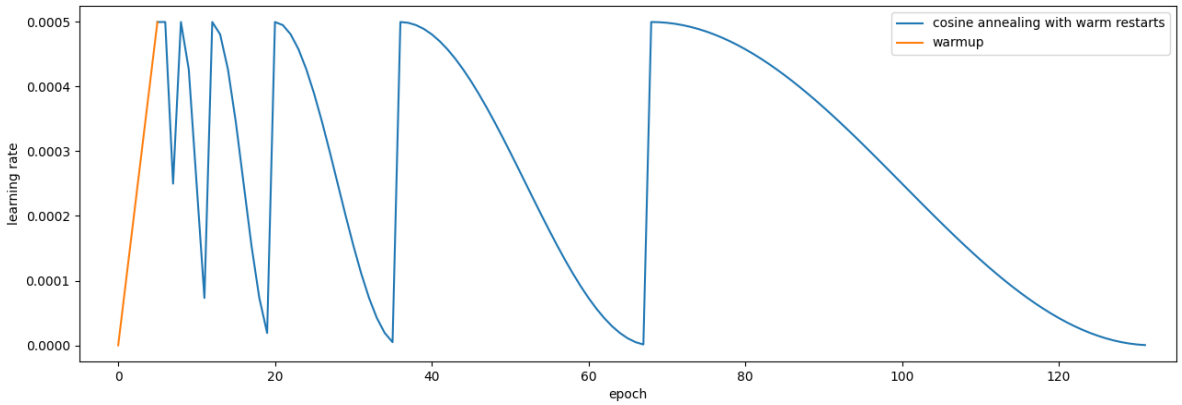


Fig. 3.3: Learning rate function used for training

3.4 Simulation Results

3.4.1 Visualizations of trajectory prediction

In this section, the trajectory prediction results are visualized and compared with the original HiVT-64.

Figure 3.4 shows an example of the visualization. For a better view, only the historical trajectory and prediction results of one agent are plotted. The orange line is the historical trajectory, the green lines are the prediction results, and the red line is the ground truth trajectory. The confidence score of each predicted trajectory is indicated by the transparency of green, and the darker represents the higher confidence score. Additionally, some metrics are plotted on the figure, where the blue arrow and pink arrow represent minFDE for K=1 and K=6. The two percentages are probabilities of two trajectories, the percentage in bold is the probability of trajectory with the highest confidence score, and the other one is the probability of trajectory with the lowest minFDE (K=6). Besides, the cyan dots indicate the position of other agents in the fifth second, and the gray polygons and arrows are the lane segments and directions. The coordinates of the Figure are in meters. The following will show the comparison of several typical scenarios.

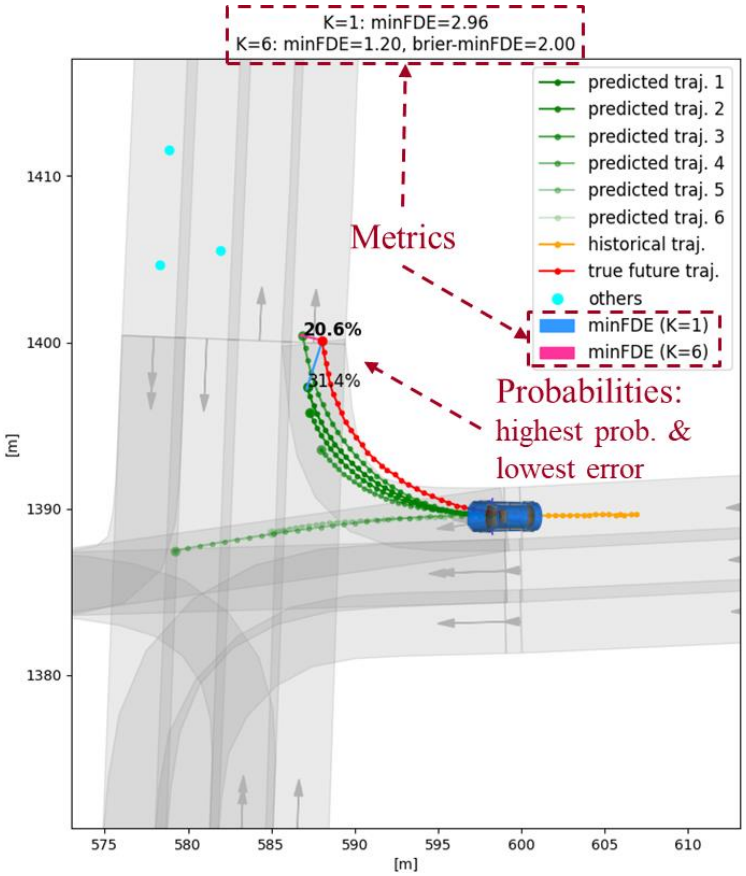
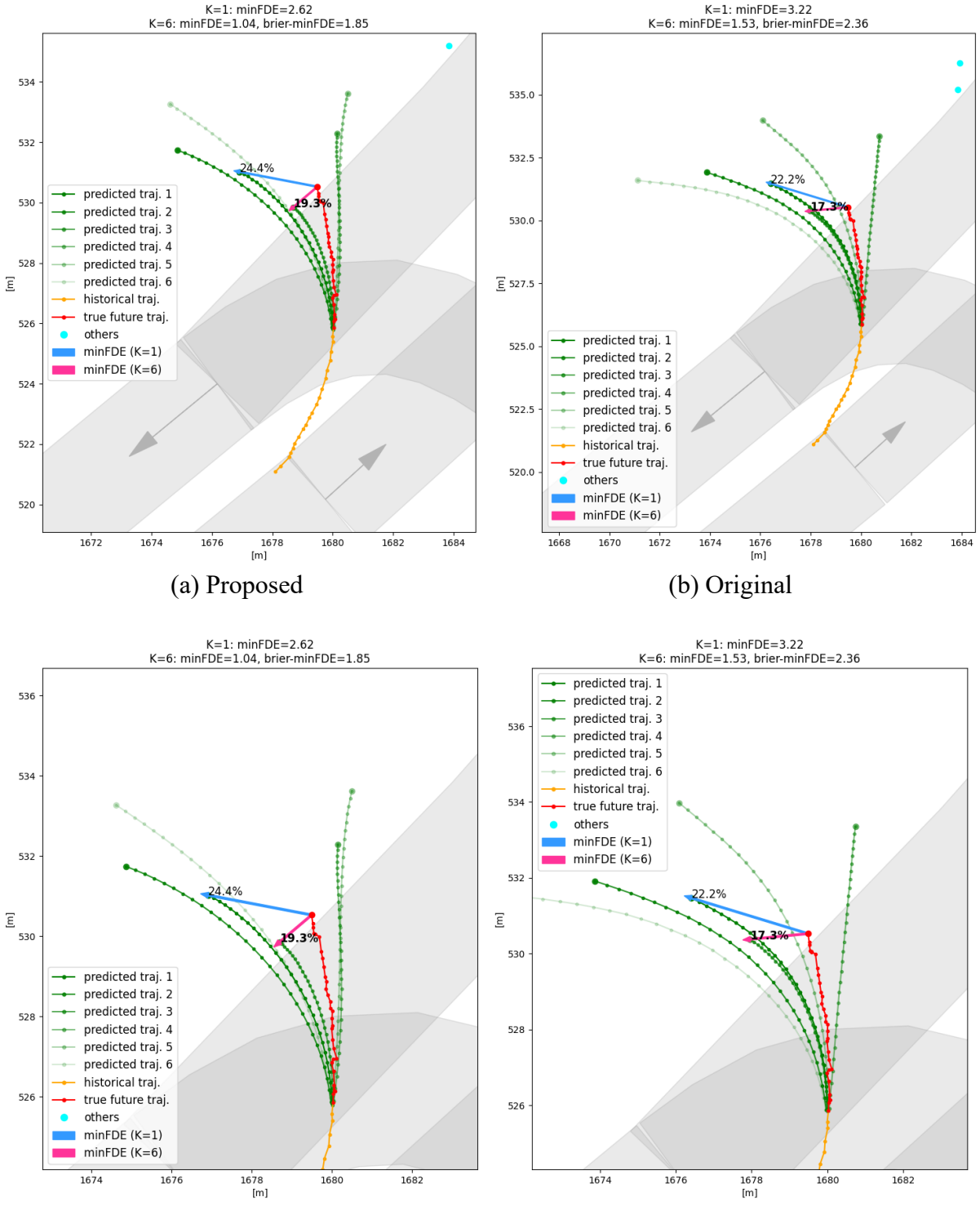


Fig. 3.4: Example of prediction results visualization

Among the various kinds of scenarios, prediction at intersections is considered to be more difficult. Figure 3.5 shows the abnormal behavior at an intersection where the vehicle turns left from the straight-ahead lane and leaves the road. Although the results of both models are not so good, the proposed model has a lower error.

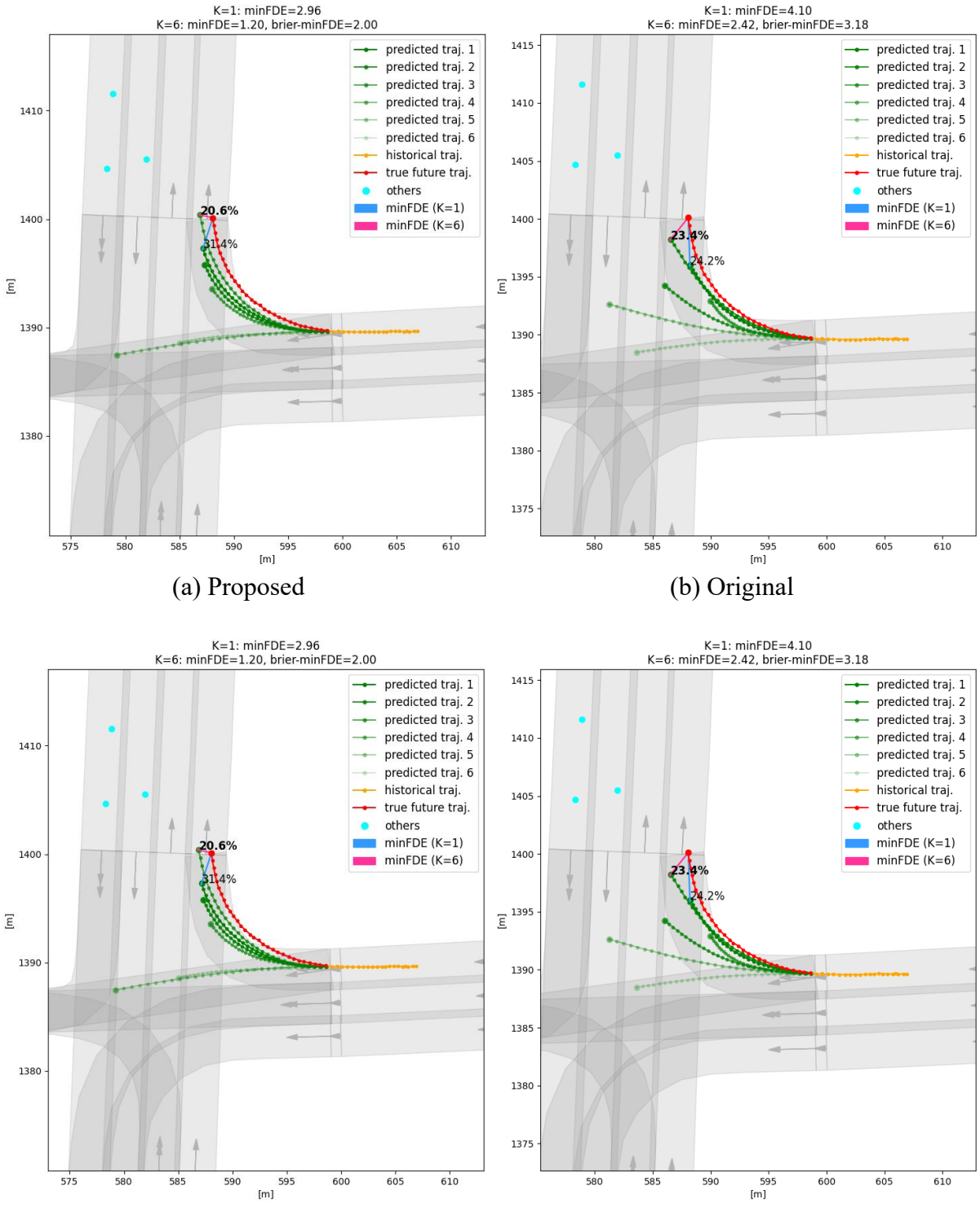


(a) Proposed (b) Original

(c) Proposed (zoom in) (d) Original (zoom in)

Fig. 3.5: Abnormal behavior (get off the road)

Figure 3.6 shows a right-turn scenario. It can be seen that the trajectories of the proposed model fit the road better. And although the original model also gets the right-turn trajectories, some trajectories are not as reasonable as the proposed model. It can be assumed that the introduction of rich lane adjacency information is beneficial for the prediction of such complex scenarios.

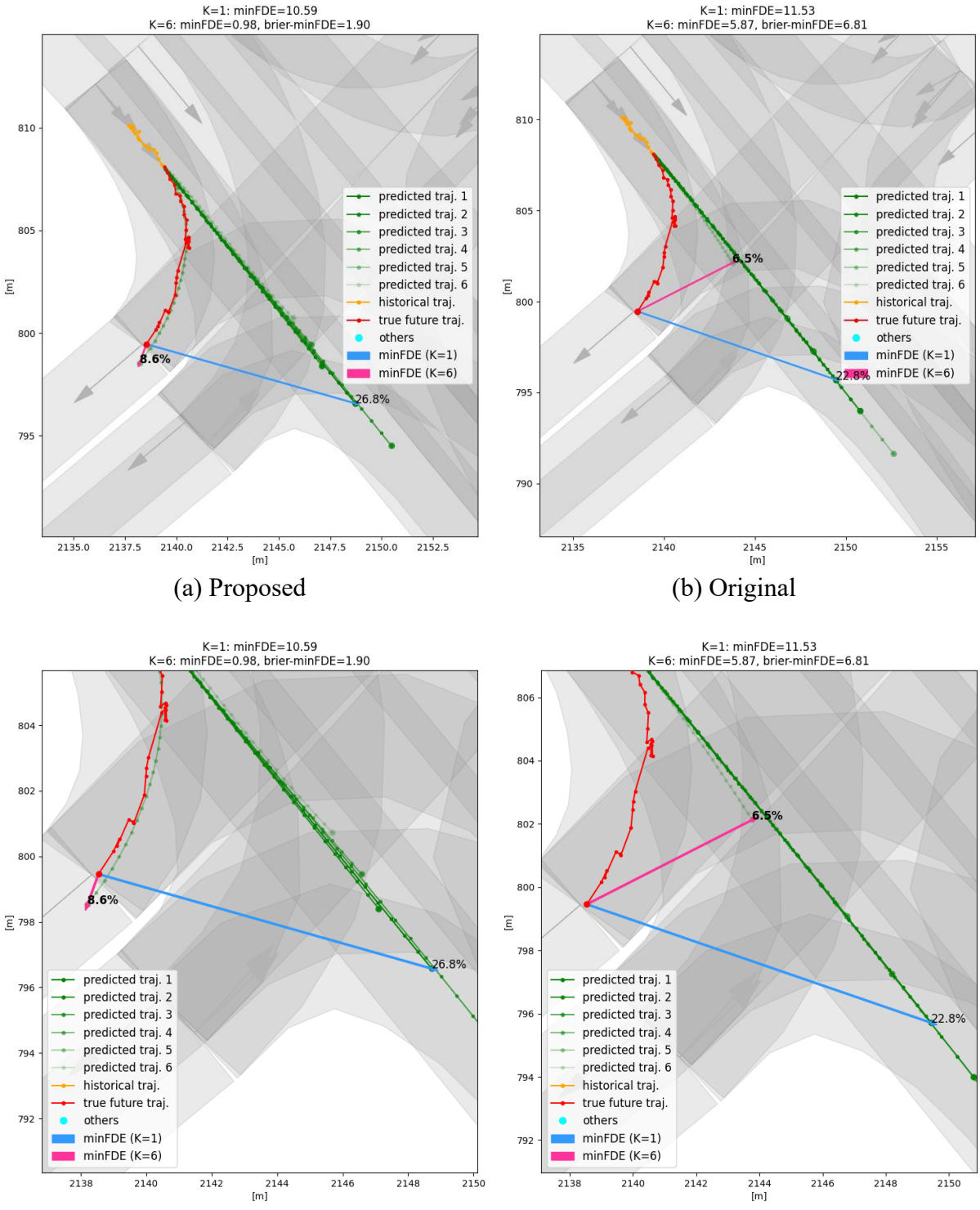


(a) Proposed (b) Original

(c) Proposed (zoom in) (d) Original (zoom in)

Fig. 3.6: Turning right (long historical trajectory)

Figure 3.7 shows another right-turn scenario, but with a shorter historical trajectory and more difficult to predict accurately. In this scenario, although the confidence score is not high, the proposed model obtains a right-turn trajectory, while the original model fails.

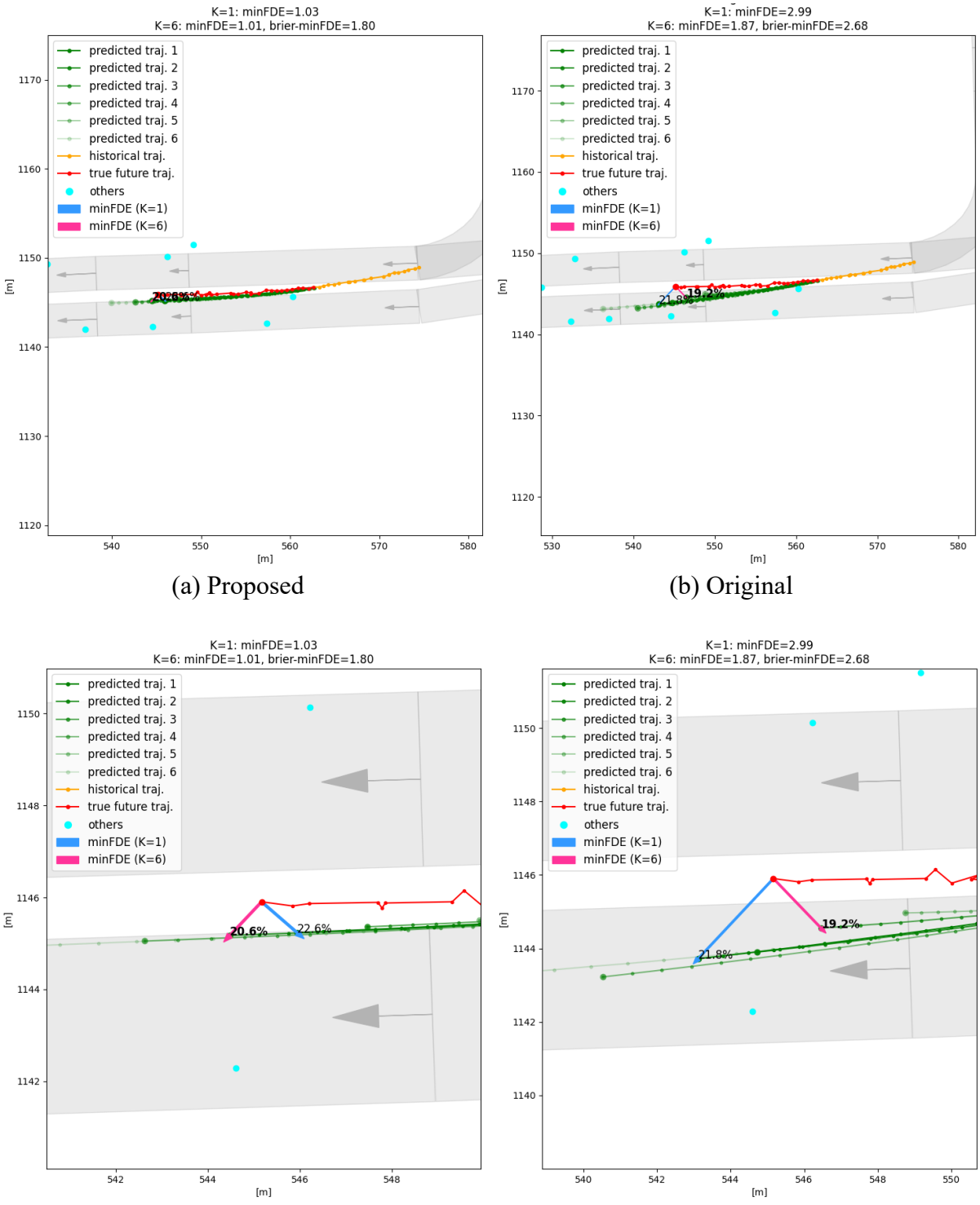


(a) Proposed (b) Original

(c) Proposed (zoom in) (b) Original (zoom in)

Fig. 3.7: Turning right (short historical trajectory)

Figure 3.8 shows a straight-driving scenario, which is easier for the model to get results with lower errors. In this scenario, maybe to avoid pedestrians on the roadside, the vehicle does not strictly follow the lane centerline. But the trajectories predicted by the original model are basically along the historical trajectory and almost collide with other agents at the end. It can be assumed that the proposed model benefits from the introduction of lane information that better handles A-A interaction.



(a) Proposed (b) Original

(c) Proposed (zoom in) (d) Original (zoom in)

Fig. 3.8: Going straight

3.4.2 Quantitative comparisons

In this section, metrics are evaluated using the official script provided by the Argoverse dataset. The most used metrics minADE, minFDE, MR, and brier-minFDE are compared with the original model. Also, for the work on model lightweight that this research focuses on in this research, the number of parameters, inference speed, and GPU memory cost is compared with the original model.

Table 3.1 and 3.2 evaluates the results of predicting a single trajectory (K=1) and 6 trajectories (K=6) on the validation and test sets, respectively. The results show that the modified model is not inferior to the original in accuracy but is even improved.

TABLE 3.1: Metrics comparison of original and improved HiVT-64 on the validation set

Model	**K=1**			**K=6**			
	minADE	**minFDE**	**MR**	**minADE**	**minFDE**	**MR**	**brier-minFDE**
Baseline	1.4293	3.1138	0.5022	0.7082	1.0757	0.1117	1.7384
Proposed	**1.3480**	**2.9234**	**0.4875**	**0.6955**	**1.0157**	**0.0984**	**1.6707**
Improve	5.69%	6.11%	2.92%	1.80%	5.57%	11.91%	3.89%

TABLE 3.2: Metrics comparison of original and improved HiVT-64 on the test set

Model	**K=1**			**K=6**			
	minADE	**minFDE**	**MR**	**minADE**	**minFDE**	**MR**	**brier-minFDE**
Baseline	1.7771	3.9306	0.5991	0.8552	1.3672	0.1625	2.0372
Proposed	**1.7161**	**3.7626**	**0.5762**	**0.8349**	**1.2654**	**0.1422**	**1.9338**
Improve	3.43%	4.28%	3.82%	2.38%	7.44%	12.49%	5.07%

Table 3.3 shows a comparison of the number of parameters, inference speed, and GPU memory cost of the model. Additionally, the lightest model in this paper is included for comparison. It can be seen that the proposed model requires significantly lower requirements for hardware than the original model with improved accuracy. This proves that the two-stage modification performed in this paper is effective.

TABLE 3.3: Specification comparison of the original and improved HiVT-64

Model	**#Param.**	**Inference Speed (fps)***	**GPU Memory Cost (MB)**			
			Local Encoder	**Global Interactor**	**Decoder**	**Others†**
Baseline	653,369	19.11	430	22	4	10
Proposed	**395,809**	**70.70**	**242**	**2**	**4**	**6**
Improve	39.42%	270.04%	43.72%	90.91%	0.00%	40.00%
Lightest	389,689	74.05	/	/	/	/

Finally, the same as the HiVT, increase the hidden size to 128, to train a larger model for comparison. Here the learning rate warmup is extended to 8 epochs, the rest remains the same. Table 3.4 shows the metrics and the comparison with other methods on the leaderboard.

* Using the same 5000 samples for inference, taking the average of three trials.

† Includes model and pre-trained weights.

TABLE 3.4: Comparison with other methods on the leaderboard (test set, K=6)

Model	minADE	minFDE	MR	brier-minFDE	#Param.
LaneGCN	0.8679	1.3640	0.1634	2.0585	3,701K
DenseTNT	0.8817	1.2815	0.1258	1.9759	1,103K
GOHOME	0.9425	1.4503	0.1048	1.9834	400K
HOME + GOHOME	0.8904	1.2919	0.0846	1.8601	5,100K
Scene Transformer	0.8026	1.2321	0.1255	1.8868	15,296K
HiVT-64	0.8552	1.3672	0.1625	2.0372	653K
HiVT-128	0.8209	1.2800	0.1482	1.9493	2,560K
Proposed	0.8349	1.2654	0.1422	1.9338	396K
Proposed-128	0.8140	1.2213	0.1326	1.8909	1,553K

Note: According to the code and pre-trained weights released by the authors of HiVT [35], the number of parameters and metrics do not match those in the paper. To ensure consistency, the results here are based on the officially released pre-trained weights but not the ones in the paper.

3.4.3 Ablation study and effectiveness validation

This section demonstrates the effect of some of the modifications on the metrics and the number of parameters through the ablation study. Metrics are evaluated only on the validation set.

The ablation studies for the decoder, encoder, and training strategy are shown in Tables 3.5 to 3.7, respectively. The results prove that these modifications are effective. And even with these complications to the model, the number of parameters is still lower than the original one.

TABLE 3.5: Ablation study on the decoder starting with the lightest model (K=6)

More layers	Sparse output w/ B-spline interp.	Residual connection	minADE	minFDE	MR	brier-minFDE	#Param.
			0.7110	1.0608	0.1091	1.7236	389,689
✓	✓		0.7063	1.0373	0.1041	1.6941	391,117
✓	✓	✓	**0.7042**	**1.0346**	**0.1027**	**1.6935**	391,117

TABLE 3.6: Ablation study on the encoder continues from Table 3.5 (K=6)

Time weight	Lane adj.	Extra embed.	Fuse edge attr.	minADE	minFDE	MR	brier-minFDE	#Param.
				0.7042	1.0346	0.1027	1.6935	391,117
✓	✓			**0.7038**	**1.0292**	0.1018	1.6860	391,521
✓	✓	✓		0.7052	1.0325	**0.1002**	1.6888	395,681
✓	✓	✓	✓	0.7041	1.0295	0.1005	**1.6852**	395,809

TABLE 3.7: Ablation study on the training strategy continues from Table 3.6

KL div. loss	LR warmup & restarts	K=1		K=6				#Param.
		minFDE	minFDE	minADE	minFDE	MR	brier-minFDE	
		1.3806	2.9902	0.7041	1.0295	0.1005	1.6852	395,809
✓		1.3687	2.9724	0.7036	1.0327	0.1013	1.6881	395,809
✓	✓	**1.3480**	**2.9234**	**0.6955**	**1.0157**	**0.0984**	**1.6707**	395,809

Starting from Table 3.6, as the number of parameters increases, the accuracy goes into a bottleneck, and some metrics even become worse. Here, it is assumed that the extra layers for embedding encoding and the change in the edge attribute fusion approach increase the training difficulty rather than reduce the feature extraction capability of the encoder. The results in Table 3.7 can also indicate that the improvement from the modification of the encoder is better exploited after the training strategy is optimized. Among them, to further confirm the effectiveness of the modified edge attribute fusion approach, a comparison experiment was carried out using the original fusion approach with the improved training strategy, and the results are shown in Table 3.8.

TABLE 3.8: Comparison between the original fusion approach and proposed

Model	K=1			K=6			
	minADE	minFDE	MR	minADE	minFDE	MR	brier-minFDE
Original	1.3732	2.9864	0.4913	0.7018	1.0280	0.1010	1.6868
Proposed	**1.3480**	**2.9234**	**0.4875**	**0.6955**	**1.0157**	**0.0984**	**1.6707**

Finally, Figure 3.10 shows the results of the time weights after training. The first step has a higher weight, and then starting from the second step, the weights gradually increase from low. This result is consistent with the hypothesis in the previous section.

```
self.time_wgt.squeeze().detach().cpu().numpy()
array([2.062761  , 0.51705956, 0.9902031 , 1.0632056 , 1.1559378 ,
       1.1959459 , 1.2953184 , 1.368692  , 1.4293575 , 1.5335826 ,
       1.616901  , 1.7310858 , 1.8082753 , 1.9473771 , 2.0934815 ,
       2.3145213 , 2.5368783 , 2.8813136 , 3.2132146 , 3.3525586 ],
      dtype=float32)
```

Fig. 3.10: Time weights

Chapter 4
Conclusions and Challenges to Future

This paper proposed an improved HiVT model with higher accuracy, prediction speed, and lower memory cost than the original one.

In chapter 2, the task of trajectory prediction, and the history of its development are introduced. The details of the HiVT model and the reasons for choosing it are highlighted. Subsequently, a detailed explanation of how the two-stage modification of the HiVT model was carried out in this research and the rationality of each modification is discussed.

In chapter 3, the simulation details were presented, and both visualization and quantitative comparisons show the proposed model outperforms the original model. And, according to the ablation study, it was shown that each of the modifications contributes to the accuracy improvement. It also proves that the two-stage modifications to the model are effective.

Nevertheless, it must be said that the target of this work is to propose a prediction network that is friendly to low-performance devices, but due to the need to keep the accuracy not decreasing much, many modifications that could improve the inference speed were eventually not adopted. Similarly, when doing accuracy compensation, there are also constraints by the inference speed, so, many modifications that would improve accuracy are not adopted.

As can be seen from the comparison with HiVT-128, the modifications in this study are more suitable for smaller models. As the number of parameters increases, the accuracy advantage becomes smaller.

Also, since motion planning does not be integrated, how much the prediction model can help motion planning has not been tested.

In addition, due to the limitations of the Argoverse dataset, including its poor semantic information, and unsmooth trajectory, which I believe also impacts the performance of the model, thus it is still unclear where the maximum performance of the model is.

Acknowledgment

In the years of the Covid-19 pandemic, many things become different and difficult. During the short two years of the master, I would like to firstly thank Professor Lee for accepting me into the lab and agreeing to start my research in the field of trajectory prediction. Through the two-year study, I gradually grew from a student who only knew how to take classes to a master's student with research ability.

I would also like to thank all my friends during the short time of less than a year in Japan. The accompany of friends in Lee lab made me feel warm even in a foreign country. Friends in my hometown also called to comfort me when I was down.

Finally, I would like to thank my parents. They allowed me to gap for more than a year after undergraduate studies and afforded me to come to Japan for graduate studies. Never asked me to be outstanding, but only hoped for me to be healthy. I hope that with the completion of this thesis, I have delivered a satisfactory answer.

References

[1] https://www.sae.org/standards/content/j3016_202104

[2] https://crashstats.nhtsa.dot.gov/Api/Public/Publication/812506#:~:text=Among%20an%20estimated%202%2C046%2C000%20drivers,2.7%25)%20of%20the%20crashes.

[3] https://www.nhtsa.gov/document/summary-report-standing-general-order-ads

[4] https://www.nhtsa.gov/document/summary-report-standing-general-order-adas-l2

[5] Werling, Moritz, et al. "Optimal Trajectory Generation for Dynamic Street Scenarios in a Frenet Frame." *2010 IEEE International Conference on Robotics and Automation*. IEEE, 2010.

[6] McNaughton, Matthew, et al. "Motion Planning for Autonomous Driving with a Conformal Spatiotemporal Lattice." *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2011.

[7] Bojarski, Mariusz, et al. "End to End Learning for Self-Driving Cars." *arXiv preprint arXiv:1604.07316* (2016).

[8] Ammoun, Samer, and Fawzi Nashashibi. "Real time trajectory prediction for collision risk estimation between vehicles." *2009 IEEE 5th International Conference on Intelligent Computer Communication and Processing*. IEEE, 2009.

[9] Phan-Minh, Tung, et al. "CoverNet: Multimodal Behavior Prediction Using Trajectory Sets." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020.

[10] Liang, Ming, et al. "Learning Lane Graph Representations for Motion Forecasting." *European Conference on Computer Vision*. Springer, Cham, 2020.

[11] Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

[12] Zhou, Zikang, et al. "HiVT: Hierarchical Vector Transformer for Multi-Agent Motion Prediction." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022.

[13] Huang, Yanjun, et al. "A Survey on Trajectory-Prediction Methods for Autonomous Driving." *IEEE Transactions on Intelligent Vehicles* (2022).

[14] Lefèvre, Stéphanie, Dizan Vasquez, and Christian Laugier. "A Survey on Motion Prediction and Risk Assessment for Intelligent Vehicles." *ROBOMECH journal* 1.1 (2014): 1-14.

[15] Park, Seong Hyeon, et al. "Sequence-to-Sequence Prediction of Vehicle Trajectory via LSTM Encoder-Decoder Architecture." *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2018.

[16] Gilles, Thomas, et al. "HOME: Heatmap Output for Future Motion Estimation." *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*. IEEE, 2021.

[17] Cui, Henggang, et al. "Deep Kinematic Models for Kinematically Feasible Vehicle Trajectory Predictions." *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020.

[18] Gao, Jiyang, et al. "VectorNet: Encoding HD Maps and Agent Dynamics from Vectorized Representation." *Proceedings of the IEEE/CVF Conference on Computer Vision and*

Pattern Recognition. 2020.
[19] Yu, Fisher, and Vladlen Koltun. "Multi-Scale Context Aggregation by Dilated Convolutions." *arXiv preprint arXiv:1511.07122* (2015).
[20] Lin, Tsung-Yi, et al. "Feature Pyramid Networks for Object Detection." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017.
[21] Da, Fang, and Yu Zhang. 'Path-Aware Graph Attention for HD Maps in Motion Prediction'. *2022 International Conference on Robotics and Automation (ICRA)*, 2022.
[22] Zhang, Chen, et al. "Technical Report for Argoverse2 Challenge 2022--Motion Forecasting Task." *arXiv preprint arXiv:2206.07934* (2022).
[23] Zhao, Hang, et al. "TNT: Target-DriveN Trajectory Prediction." *Conference on Robot Learning*. PMLR, 2021.
[24] Bochkovskiy, Alexey, Chien-Yao Wang, and Hong-Yuan Mark Liao. "YOLOv4: Optimal Speed and Accuracy of Object Detection." *arXiv preprint arXiv:2004.10934* (2020).
[25] Gu, Junru, Chen Sun, and Hang Zhao. "DenseTNT: End-to-End Trajectory Prediction from Dense Goal Sets." *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021.
[26] Dosovitskiy, Alexey, et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale." *arXiv preprint arXiv:2010.11929* (2020).
[27] Jiquan Ngiam, et al. "Scene Transformer: A Unified Architecture for Predicting Future Trajectories of Multiple Agents." *International Conference on Learning Representations*. 2022.
[28] Gilles, Thomas, et al. "GOHOME: Graph-Oriented Heatmap Output for Future Motion Estimation." *2022 International Conference on Robotics and Automation (ICRA)*. IEEE Press, 2022.
[29] He, Kaiming, et al. "Deep Residual Learning for Image Recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.
[30] Huang, Xiao Shi, et al. "Improving Transformer Optimization Through Better Initialization." *International Conference on Machine Learning*. PMLR, 2020.
[31] Loshchilov, Ilya, and Frank Hutter. "SGDR: Stochastic Gradient Descent with Warm Restarts." *arXiv preprint arXiv:1608.03983* (2016).
[32] Chang, Ming-Fang, et al. "Argoverse: 3D Tracking and Forecasting with Rich Maps." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019.
[33] Brier, Glenn W. "Verification of Forecasts Expressed in Terms of Probability." *Monthly weather review* 78.1 (1950): 1-3.
[34] Loshchilov, Ilya, and Frank Hutter. "Decoupled Weight Decay Regularization." *arXiv preprint arXiv:1711.05101* (2017).
[35] https://github.com/ZikangZhou/HiVT

Appendix

Main code with model, training, and evaluation. Code for data processing and visualization etc. is not included.

hivt.py

```
import torch
import torch.nn as nn
from scipy.interpolate import splrep, splev
import numpy as np
from utils import TemporalData

from losses import LaplaceNLLLoss, KLDivLoss
from metrics import ADE, FDE, BFDE, MR, Loss, ClsLoss, RegLoss
from models import LocalEncoderMod, GlobalInteractorMod, DecoderMod


class HiVT(nn.Module):

    def __init__(self,
                 historical_steps: int,
                 future_steps: int,
                 num_modes: int,
                 node_dim: int,
                 edge_dim: int,
                 embed_dim: int,
                 num_heads: int,
                 dropout: float,
                 num_temporal_layers: int,
                 local_radius: float,
                 lr: float,
                 weight_decay: float,
                 per_sec: bool = False,
                 infer: bool = False,
                 **kwargs) -> None:
        super(HiVT, self).__init__()
        self.historical_steps = historical_steps
        self.future_steps = future_steps
        self.num_modes = num_modes
        self.lr = lr
        self.weight_decay = weight_decay
        uncertain = not infer
        self.infer = infer

        self.local_encoder = LocalEncoderMod(historical_steps=historical_steps,
                                             node_dim=node_dim,
                                             edge_dim=edge_dim,
```

```
                                        embed_dim=embed_dim,
                                        num_heads=num_heads,
                                        dropout=dropout,
                                        num_temporal_layers=num_temporal_layers,
                                        local_radius=local_radius)
        self.global_interactor = GlobalInteractorMod(historical_steps=historical_steps,
                                                     embed_dim=embed_dim,
                                                     edge_dim=edge_dim,
                                                     num_modes=num_modes,
                                                     num_heads=num_heads,
                                                     dropout=dropout)
        self.decoder = DecoderMod(embed_dim=embed_dim,
                                  future_steps=future_steps,
                                  num_modes=num_modes,
                                  uncertain=uncertain)
        self.reg_loss = LaplaceNLLLoss(reduction='mean')
        self.cls_loss = KLDivLoss(reduction='batchmean')

        self.minADE = ADE()
        self.minFDE = FDE()
        self.MR = MR()
        self.minADE1 = ADE()
        self.minFDE1 = FDE()
        self.MR1 = MR()
        self.b_minFDE = BFDE()
        self.cls_loss_val = ClsLoss()
        self.reg_loss_val = RegLoss()
        self.loss_val = Loss()
        self.per_sec = per_sec
        if per_sec:  # sparse output + interpolation
            mask = [False if (i + 1) % 10 != 0 else True for i in range(self.future_steps)]
            self.mask = torch.tensor(mask)
            self.t0 = np.arange(0, future_steps + 1, 10, dtype=np.float64)
            self.t = np.arange(0, future_steps + 1, 1, dtype=np.float64)
            self.start = np.asarray([[0, 0]], dtype=np.float32)

    def forward(self, data: TemporalData):
        sin_vals = torch.sin(data['rotate_angles']).view(-1, 1, 1)
        cos_vals = torch.cos(data['rotate_angles']).view(-1, 1, 1)
        rotate_mat = torch.cat((torch.hstack((cos_vals, sin_vals)),
                                torch.hstack((-sin_vals, cos_vals))), dim=-1)
        if data.y is not None:
            data.y = torch.bmm(data.y, rotate_mat)
        data['rotate_mat'] = rotate_mat

        local_embed = self.local_encoder(data=data)
        global_embed = self.global_interactor(data=data, local_embed=local_embed)
        y_hat, pi = self.decoder(local_embed=local_embed, global_embed=global_embed)
        # y_hat: [6, N, 30, 4]
```

```
        # y_hat[:, :, :, :2]: position
        # y_hat[:, :, :, 2:]: uncertainty (sigma of laplace dist.)
        # pi: [N, 6], mixing coefficient of GMM (prob. of each mode)

        agent_idx = data['agent_index']
        y_hat_agent = y_hat.detach()[:, agent_idx, :, :2]  # [num_modes, batch_size, 30, 2]

        if not self.infer:
            reg_mask = ~data['padding_mask'][:, self.historical_steps:]
            y = data.y
            if self.per_sec:
                reg_mask = reg_mask[:, self.mask]
                y = y[:, self.mask]
                y_hat_agent = self.traj_interp(y_hat_agent)

            l2_norm = (torch.norm(y_hat[:, :, :, :2] - y, p=2, dim=-1) *
reg_mask).sum(dim=-1)  # [F, N]
            valid_steps = reg_mask.sum(dim=-1)
            cls_mask = valid_steps > 0
            best_mode = l2_norm.argmin(dim=0)
            y_hat_best = y_hat[best_mode, torch.arange(data.num_nodes)]

            loss = {}
            loss['reg_loss'] = self.reg_loss(y_hat_best[reg_mask], y[reg_mask])
            soft_target = torch.softmax(-l2_norm[:, cls_mask] / valid_steps[cls_mask],
dim=0).detach().t()
            loss['cls_loss'] = self.cls_loss(pi[cls_mask], soft_target)
            loss['loss'] = loss['reg_loss'] + loss['cls_loss']

            self.update_metrics(y_hat_agent, data.y[agent_idx], pi.detach()[agent_idx],
torch.arange(data.num_graphs), loss)
            return loss
        else:
            if self.per_sec:
                y_hat_agent = self.traj_interp(y_hat_agent)

            sin_vals = torch.sin(data['theta']).unsqueeze(-1)
            cos_vals = torch.cos(data['theta']).unsqueeze(-1)
            global_rotate_mat = torch.cat((torch.hstack((cos_vals, -sin_vals)).unsqueeze(-
1),
                                           torch.hstack((sin_vals, cos_vals)).unsqueeze(-
1)), dim=-1)
            data['global_rotate_mat'] = global_rotate_mat
            position, prob = self.post_process(y_hat_agent, pi.detach()[agent_idx],
                                               rotate_mat[agent_idx].transpose(1, 2),
                                               data['positions'][agent_idx][:,
19].unsqueeze(1),
                                               global_rotate_mat,
data['origin'].unsqueeze(1))
```

```
            return position, prob


    def update_metrics(self, y_hat_agent, y_agent, conf, row_idx, loss):
        best_mode_agent = torch.norm(y_hat_agent[:, :, -1] - y_agent[:, -1], p=2, dim=-
1).argmin(dim=0)
        y_hat_best_agent = y_hat_agent[best_mode_agent, row_idx]  # [batch_size, 30, 2]
        probs = torch.softmax(conf, dim=1)[row_idx, best_mode_agent]
        y_hat_max_prob_agent = y_hat_agent[conf.argmax(dim=1), row_idx]  # [batch_size, 30,
2]

        # update metrics (k=6)
        self.minADE.update(y_hat_best_agent, y_agent)
        self.minFDE.update(y_hat_best_agent, y_agent)
        self.MR.update(y_hat_best_agent, y_agent)
        self.b_minFDE.update(y_hat_best_agent, y_agent, probs)

        # update metrics (k=1)
        self.minADE1.update(y_hat_max_prob_agent, y_agent)
        self.minFDE1.update(y_hat_max_prob_agent, y_agent)
        self.MR1.update(y_hat_max_prob_agent, y_agent)

        # update loss value
        self.cls_loss_val.update(loss["cls_loss"].detach())
        self.reg_loss_val.update(loss["reg_loss"].detach())
        self.loss_val.update(loss["loss"].detach())


    def post_process(self, y_hat_agent: torch.Tensor, conf: torch.Tensor,
                     local_rotate_mat: torch.Tensor, local_origin: torch.Tensor,
                     global_rotate_mat: torch.Tensor, global_origin: torch.Tensor):
        local_positions = [(torch.bmm(y_hat, local_rotate_mat) + local_origin)
                           for y_hat in y_hat_agent]
        positions = torch.cat([(torch.bmm(y_hat, global_rotate_mat) +
global_origin).unsqueeze(1)
                               for y_hat in local_positions],
                              dim=1)
        probs = torch.softmax(conf, dim=1)
        probs_sort, idx = torch.sort(probs, dim=1, descending=True)
        positions_sorted = torch.stack([positions[i][j] for i, j in enumerate(idx)])  #
[batch_size, 30, 2]
        return positions_sorted.cpu().numpy(), probs_sort.cpu().numpy()

    def traj_interp(self, y_hat_agent):
        interp_y = []
        device_ = y_hat_agent.device
        y_hat_agent = y_hat_agent.detach().cpu().numpy()
        start = np.broadcast_to(self.start, y_hat_agent[:, :, :1, :].shape)
        y_hat_agent_arr = np.concatenate((start, y_hat_agent), axis=2, dtype=np.float64)
```

```
        for mode in y_hat_agent_arr:
            tmp = []
            for xy0 in mode:
                xnew = splev(self.t, splrep(self.t0, xy0[:, 0]))[1:, np.newaxis]
                ynew = splev(self.t, splrep(self.t0, xy0[:, 1]))[1:, np.newaxis]
                tmp.append(np.hstack((xnew, ynew)))
            interp_y.append(np.asarray(tmp, dtype=np.float32))
        return torch.from_numpy(np.asarray(interp_y)).to(device_)


    @staticmethod
    def add_model_specific_args(parent_parser):
        parser = parent_parser.add_argument_group('HiVT')
        parser.add_argument('--historical_steps', type=int, default=20)
        parser.add_argument('--future_steps', type=int, default=30)
        parser.add_argument('--num_modes', type=int, default=6)
        parser.add_argument('--node_dim', type=int, default=2)
        parser.add_argument('--edge_dim', type=int, default=2)
        parser.add_argument('--embed_dim', type=int, required=True)
        parser.add_argument('--num_heads', type=int, default=8)
        parser.add_argument('--dropout', type=float, default=0.1)
        parser.add_argument('--num_temporal_layers', type=int, default=4)
        parser.add_argument('--num_global_layers', type=int, default=3)
        parser.add_argument('--local_radius', type=float, default=50)
        parser.add_argument('--lr', type=float, default=5e-4)
        parser.add_argument('--weight_decay', type=float, default=1e-4)
        return parent_parser
```

local_encoder_mod.py

```
from typing import Optional, Tuple, List
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn.conv import MessagePassing
from torch_geometric.typing import Adj, OptTensor, Size
from torch_geometric.utils import softmax, subgraph
from utils import DistanceDropEdge, TemporalData, init_weights


class LocalEncoderMod(nn.Module):

    def __init__(self,
                 historical_steps: int,
                 node_dim: int,
                 edge_dim: int,
                 embed_dim: int,
                 num_heads: int = 8,
```

```
                 dropout: float = 0.1,
                 num_temporal_layers: int = 4,
                 local_radius: float = 50) -> None:
        super(LocalEncoderMod, self).__init__()
        self.historical_steps = historical_steps

        self.drop_edge = DistanceDropEdge(local_radius)
        self.aa_encoder = AAEncoder(historical_steps=historical_steps,
                                    edge_dim=edge_dim,
                                    embed_dim=embed_dim,
                                    num_heads=num_heads,
                                    dropout=dropout)
        self.temporal_encoder = TemporalEncoder(historical_steps=historical_steps,
                                                embed_dim=embed_dim,
                                                num_heads=num_heads,
                                                dropout=dropout,
                                                num_layers=num_temporal_layers)
        self.al_encoder = ALEncoder(node_dim=node_dim,
                                    edge_dim=edge_dim,
                                    embed_dim=embed_dim,
                                    num_heads=num_heads,
                                    dropout=dropout)
        self.agt_embed = AgtEmbed(in_channel=node_dim, out_channel=embed_dim)
        self.bos_token = nn.Parameter(torch.Tensor(historical_steps, embed_dim))

    def forward(self, data: TemporalData) -> torch.Tensor:
        # Temporal encoding
        out = self.agt_embed(torch.bmm(data.x, data['rotate_mat']))
        out = torch.where(data['bos_mask'].unsqueeze(-1), self.bos_token, out).transpose(0,
1)
        out = self.temporal_encoder(x=out,
                                    padding_mask=data['padding_mask'][:, :self.historical_s
teps])
        # L2A
        edge_index, edge_attr = self.drop_edge(data['lane_actor_index'],
                                               data['lane_actor_vectors'])
        out = self.al_encoder(x=(data['lane_vectors'], out),
                              edge_index=edge_index,
                              edge_attr=edge_attr,
                              is_intersections=data['is_intersections'],
                              turn_directions=data['turn_directions'],
                              traffic_controls=data['traffic_controls'],
                              left_nbrs=data['left_nbrs'],
                              right_nbrs=data['right_nbrs'],
                              rotate_mat=data['rotate_mat'])
        # A2A
        edge_index, _ = subgraph(subset=~data['padding_mask'][:, self.historical_steps -
1],
                                 edge_index=data.edge_index)
```

```
        edge_attr = data['positions'][edge_index[0], self.historical_steps - 1] -
data['positions'][
            edge_index[1], self.historical_steps - 1]
        edge_index, edge_attr = self.drop_edge(edge_index, edge_attr)
        out = self.aa_encoder(x=out,
                              edge_index=edge_index,
                              edge_attr=edge_attr,
                              rotate_mat=data['rotate_mat'])
        return out


class AgtEmbed(nn.Module):
    '''Agent Encoder'''

    def __init__(self, in_channel: int, out_channel: int) -> None:
        super(AgtEmbed, self).__init__()
        self.embed = nn.Sequential(
            nn.Linear(in_channel, out_channel),
            nn.LayerNorm(out_channel),
            nn.ReLU(inplace=True),
            nn.Linear(out_channel, out_channel, bias=False),
            nn.LayerNorm(out_channel),
            nn.ReLU(inplace=True),
            nn.Linear(out_channel, out_channel, bias=False),
            nn.LayerNorm(out_channel))
        self.apply(init_weights)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.embed(x)


class LocalNbrEmbed(nn.Module):
    '''Fuse Edge Attributes to Neighbor'''

    def __init__(self, in_channels: List[int], out_channel: int) -> None:
        super(LocalNbrEmbed, self).__init__()

        self.agt_emb = nn.Sequential(
            nn.Linear(in_channels[0], out_channel, bias=False),
            nn.LayerNorm(out_channel),
            nn.ReLU(inplace=True),
            nn.Linear(out_channel, out_channel, bias=False))
        self.edge_emb = nn.Sequential(
            nn.Linear(in_channels[1], out_channel),
            nn.LayerNorm(out_channel),
            nn.ReLU(inplace=True),
            nn.Linear(out_channel, out_channel, bias=False))
        self.aggr_embed = nn.Sequential(
            nn.LayerNorm(out_channel),
```

```
            nn.ReLU(inplace=True),
            nn.Linear(out_channel, out_channel, bias=False),
            nn.LayerNorm(out_channel))
        self.apply(init_weights)

    def forward(self, continuous_inputs: List[torch.Tensor]) -> torch.Tensor:
        out = self.agt_emb(continuous_inputs[0]) + self.edge_emb(continuous_inputs[1])
        return self.aggr_embed(out)


class LaneEmbed(nn.Module):
    '''Lane Encoder'''

    def __init__(self, in_channel: int, e_in_channel: int, c_in_channels: List[int],
                 out_channel: int) -> None:
        super(LaneEmbed, self).__init__()
        self.embed = nn.Sequential(
            nn.Linear(in_channel, out_channel),
            nn.LayerNorm(out_channel),
            nn.ReLU(inplace=True),
            nn.Linear(out_channel, out_channel, bias=False),
            nn.LayerNorm(out_channel),
            nn.ReLU(inplace=True),
            nn.Linear(out_channel, out_channel, bias=False))
        self.edge = nn.Sequential(
            nn.Linear(e_in_channel, out_channel),
            nn.LayerNorm(out_channel),
            nn.ReLU(inplace=True),
            nn.Linear(out_channel, out_channel, bias=False))
        self.aggr_embed = nn.Sequential(
            nn.LayerNorm(out_channel),
            nn.ReLU(inplace=True),
            nn.Linear(out_channel, out_channel, bias=False),
            nn.LayerNorm(out_channel))
        self.categorical_embed = nn.ModuleList(
            [nn.Embedding(c_in_channel, out_channel) for c_in_channel in c_in_channels])
        self.apply(init_weights)

    def forward(self,
                continuous_inputs: List[torch.Tensor],
                categorical_inputs: List[torch.Tensor] = None) -> torch.Tensor:
        output = self.embed(continuous_inputs[0]) + self.edge(continuous_inputs[1])
        for i in range(len(categorical_inputs)):
            output += self.categorical_embed[i](categorical_inputs[i])
        return self.aggr_embed(output)


class AAEncoder(MessagePassing):
```

```
    def __init__(self,
                 historical_steps: int,
                 edge_dim: int,
                 embed_dim: int,
                 num_heads: int = 8,
                 dropout: float = 0.1,
                 parallel: bool = False,
                 **kwargs) -> None:
        super(AAEncoder, self).__init__(aggr='add', node_dim=0, **kwargs)
        self.historical_steps = historical_steps
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads
        self.scale = self.head_dim**0.5
        self.parallel = parallel

        self.nbr_embed = LocalNbrEmbed(in_channels=[embed_dim, edge_dim],
                                                out_channel=embed_dim)
        self.lin_q = nn.Linear(embed_dim, embed_dim, bias=False)
        self.lin_k = nn.Linear(embed_dim, embed_dim, bias=False)
        self.lin_v = nn.Linear(embed_dim, embed_dim, bias=False)
        self.lin_self = nn.Linear(embed_dim, embed_dim, bias=False)
        self.attn_drop = nn.Dropout(dropout)
        self.lin_ih = nn.Linear(embed_dim, embed_dim, bias=False)
        self.lin_hh = nn.Linear(embed_dim, embed_dim, bias=False)
        self.out_proj = nn.Linear(embed_dim, embed_dim)
        self.proj_drop = nn.Dropout(dropout)
        self.norm1 = nn.LayerNorm(embed_dim)
        self.norm2 = nn.LayerNorm(embed_dim)
        self.mlp = nn.Sequential(nn.Linear(embed_dim, embed_dim * 2),
nn.ReLU(inplace=True),
                                 nn.Dropout(dropout), nn.Linear(embed_dim * 2, embed_dim),
                                 nn.Dropout(dropout))
        self.apply(init_weights)

    def forward(self,
                x: torch.Tensor,
                edge_index: Adj,
                edge_attr: torch.Tensor,
                rotate_mat: Optional[torch.Tensor] = None,
                size: Size = None) -> torch.Tensor:
        x = x + self._mha_block(self.norm1(x), x, edge_index, edge_attr, rotate_mat, size)
        x = x + self._ff_block(self.norm2(x))
        return x

    def message(self, edge_index: Adj, center_embed_i: torch.Tensor, x_j: torch.Tensor,
                edge_attr: torch.Tensor, rotate_mat: Optional[torch.Tensor], index:
torch.Tensor,
                ptr: OptTensor, size_i: Optional[int]) -> torch.Tensor:
```

```
        center_rotate_mat = rotate_mat[edge_index[1]]
        nbr_embed = self.nbr_embed(
            [x_j, torch.bmm(edge_attr.unsqueeze(-2), center_rotate_mat).squeeze(-2)])
        query = self.lin_q(center_embed_i).view(-1, self.num_heads, self.head_dim)
        key = self.lin_k(nbr_embed).view(-1, self.num_heads, self.head_dim)
        value = self.lin_v(nbr_embed).view(-1, self.num_heads, self.head_dim)
        alpha = torch.sum(query * key, dim=-1, keepdim=True) / self.scale
        alpha = self.attn_drop(softmax(alpha, index, ptr, size_i))
        return value * alpha

    def update(self, inputs: torch.Tensor, center_embed: torch.Tensor) -> torch.Tensor:
        inputs = inputs.view(-1, self.embed_dim)
        gate = torch.sigmoid(self.lin_ih(inputs) + self.lin_hh(center_embed))
        return inputs + gate * (self.lin_self(center_embed) - inputs)

    def _mha_block(self, center_embed: torch.Tensor, x: torch.Tensor, edge_index: Adj,
                   edge_attr: torch.Tensor, rotate_mat: Optional[torch.Tensor],
                   size: Size) -> torch.Tensor:
        """Multi-head attention block"""
        center_embed = self.out_proj(
            self.propagate(edge_index=edge_index,
                           x=x,
                           center_embed=center_embed,
                           edge_attr=edge_attr,
                           rotate_mat=rotate_mat,
                           size=size))
        return self.proj_drop(center_embed)

    def _ff_block(self, x: torch.Tensor) -> torch.Tensor:
        return self.mlp(x)


class TemporalEncoder(nn.Module):

    def __init__(self,
                 historical_steps: int,
                 embed_dim: int,
                 num_heads: int = 8,
                 num_layers: int = 4,
                 dropout: float = 0.1) -> None:
        super(TemporalEncoder, self).__init__()
        self.time_wgt = nn.Parameter(torch.ones(historical_steps, 1, 1))
        encoder_layer = TemporalEncoderLayer(embed_dim=embed_dim,
                                             num_heads=num_heads,
                                             dropout=dropout)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer=encoder_layer,
                                                         num_layers=num_layers,
                                                         norm=nn.LayerNorm(embed_dim))
        self.padding_token = nn.Parameter(torch.Tensor(historical_steps, 1, embed_dim))
```

```
        self.cls_token = nn.Parameter(torch.Tensor(1, 1, embed_dim))
        self.pos_embed = nn.Parameter(torch.Tensor(historical_steps + 1, 1, embed_dim))
        attn_mask = self.generate_square_subsequent_mask(historical_steps + 1)
        self.register_buffer('attn_mask', attn_mask)
        nn.init.normal_(self.padding_token, mean=0., std=.02)
        nn.init.normal_(self.cls_token, mean=0., std=.02)
        nn.init.normal_(self.pos_embed, mean=0., std=.02)
        self.apply(init_weights)

    def forward(self, x: torch.Tensor, padding_mask: torch.Tensor) -> torch.Tensor:
        x = torch.where(padding_mask.t().unsqueeze(-1), self.padding_token,
                        x * self.time_wgt.expand(-1, x.shape[1], -1))
        x = torch.cat((x, self.cls_token.expand(-1, x.shape[1], -1)), dim=0)
        x = x + self.pos_embed
        out = self.transformer_encoder(src=x, mask=self.attn_mask,
src_key_padding_mask=None)
        return out[-1]  # [N, D]

    @staticmethod
    def generate_square_subsequent_mask(seq_len: int) -> torch.Tensor:
        mask = (torch.triu(torch.ones(seq_len, seq_len)) == 1).transpose(0, 1)
        mask = mask.float().masked_fill(mask == 0,
                                        float('-inf')).masked_fill(mask == 1, float(0.0))
        return mask


class TemporalEncoderLayer(nn.Module):

    def __init__(self, embed_dim: int, num_heads: int = 8, dropout: float = 0.1) -> None:
        super(TemporalEncoderLayer, self).__init__()
        self.self_attn = nn.MultiheadAttention(embed_dim=embed_dim,
                                               num_heads=num_heads,
                                               dropout=dropout)
        self.linear1 = nn.Linear(embed_dim, embed_dim * 2)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(embed_dim * 2, embed_dim)
        self.norm1 = nn.LayerNorm(embed_dim)
        self.norm2 = nn.LayerNorm(embed_dim)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self,
                src: torch.Tensor,
                src_mask: Optional[torch.Tensor] = None,
                src_key_padding_mask: Optional[torch.Tensor] = None) -> torch.Tensor:
        x = src
        x = x + self._sa_block(self.norm1(x), src_mask, src_key_padding_mask)
        x = x + self._ff_block(self.norm2(x))
        return x
```

```
    def _sa_block(self, x: torch.Tensor, attn_mask: Optional[torch.Tensor],
                  key_padding_mask: Optional[torch.Tensor]) -> torch.Tensor:
        x = self.self_attn(x,
                           x,
                           x,
                           attn_mask=attn_mask,
                           key_padding_mask=key_padding_mask,
                           need_weights=False)[0]
        return self.dropout1(x)

    def _ff_block(self, x: torch.Tensor) -> torch.Tensor:
        x = self.linear2(self.dropout(F.relu_(self.linear1(x))))
        return self.dropout2(x)


class ALEncoder(MessagePassing):

    def __init__(self,
                 node_dim: int,
                 edge_dim: int,
                 embed_dim: int,
                 num_heads: int = 8,
                 dropout: float = 0.1,
                 **kwargs) -> None:
        super(ALEncoder, self).__init__(aggr='add', node_dim=0, **kwargs)
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads
        self.scale = self.head_dim**0.5

        self.lane_embed = LaneEmbed(in_channel=node_dim,
                                    e_in_channel=edge_dim,
                                    c_in_channels=[2, 3, 2, 3, 3],
                                    out_channel=embed_dim)
        self.lin_q = nn.Linear(embed_dim, embed_dim, bias=False)
        self.lin_k = nn.Linear(embed_dim, embed_dim, bias=False)
        self.lin_v = nn.Linear(embed_dim, embed_dim, bias=False)
        self.lin_self = nn.Linear(embed_dim, embed_dim, bias=False)
        self.attn_drop = nn.Dropout(dropout)
        self.lin_ih = nn.Linear(embed_dim, embed_dim, bias=False)
        self.lin_hh = nn.Linear(embed_dim, embed_dim, bias=False)
        self.out_proj = nn.Linear(embed_dim, embed_dim)
        self.proj_drop = nn.Dropout(dropout)
        self.norm1 = nn.LayerNorm(embed_dim)
        self.norm2 = nn.LayerNorm(embed_dim)
        self.mlp = nn.Sequential(
            nn.Linear(embed_dim, embed_dim * 2),
            nn.ReLU(inplace=True),
```

```
            nn.Dropout(dropout),
            nn.Linear(embed_dim * 2, embed_dim),
            nn.Dropout(dropout))
        self.apply(init_weights)

    def forward(self,
                x: Tuple[torch.Tensor, torch.Tensor],
                edge_index: Adj,
                edge_attr: torch.Tensor,
                is_intersections: torch.Tensor,
                turn_directions: torch.Tensor,
                traffic_controls: torch.Tensor,
                left_nbrs: torch.Tensor,
                right_nbrs: torch.Tensor,
                rotate_mat: Optional[torch.Tensor] = None,
                size: Size = None) -> torch.Tensor:
        x_lane, x_actor = x
        x_actor = x_actor + self._mha_block(self.norm1(x_actor), x_lane, edge_index,
edge_attr,
                                            is_intersections.long(),
turn_directions.long(),
                                            traffic_controls.long(),
                                            left_nbrs.clamp_(0, 2).long(),
                                            right_nbrs.clamp_(0, 2).long(), rotate_mat,
size)
        x_actor = x_actor + self._ff_block(self.norm2(x_actor))
        return x_actor

    def message(self, edge_index: Adj, x_i: torch.Tensor, x_j: torch.Tensor,
                edge_attr: torch.Tensor, is_intersections_j: torch.Tensor,
                turn_directions_j: torch.Tensor, traffic_controls_j: torch.Tensor,
                left_nbrs_j: torch.Tensor, right_nbrs_j: torch.Tensor,
                rotate_mat: Optional[torch.Tensor], index: torch.Tensor, ptr: OptTensor,
                size_i: Optional[int]) -> torch.Tensor:
        rotate_mat = rotate_mat[edge_index[1]]
        x_j = self.lane_embed(
            [torch.bmm(x_j.unsqueeze(-2), rotate_mat).squeeze(-2),
             torch.bmm(edge_attr.unsqueeze(-2), rotate_mat).squeeze(-2)],
            [is_intersections_j, turn_directions_j, traffic_controls_j, left_nbrs_j,
right_nbrs_j])
        query = self.lin_q(x_i).view(-1, self.num_heads, self.head_dim)
        key = self.lin_k(x_j).view(-1, self.num_heads, self.head_dim)
        value = self.lin_v(x_j).view(-1, self.num_heads, self.head_dim)
        alpha = torch.sum(query * key, dim=-1, keepdim=True) / self.scale
        alpha = self.attn_drop(softmax(alpha, index, ptr, size_i))
        return value * alpha

    def update(self, inputs: torch.Tensor, x: torch.Tensor) -> torch.Tensor:
        x_actor = x[1]
```

```
        inputs = inputs.view(-1, self.embed_dim)
        gate = torch.sigmoid(self.lin_ih(inputs) + self.lin_hh(x_actor))
        return inputs + gate * (self.lin_self(x_actor) - inputs)

    def _mha_block(self, x_actor: torch.Tensor, x_lane: torch.Tensor, edge_index: Adj,
                   edge_attr: torch.Tensor, is_intersections: torch.Tensor,
                   turn_directions: torch.Tensor, traffic_controls: torch.Tensor,
                   left_nbrs: torch.Tensor, right_nbrs: torch.Tensor,
                   rotate_mat: Optional[torch.Tensor], size: Size) -> torch.Tensor:
        x_actor = self.out_proj(
            self.propagate(edge_index=edge_index,
                           x=(x_lane, x_actor),
                           edge_attr=edge_attr,
                           is_intersections=is_intersections,
                           turn_directions=turn_directions,
                           traffic_controls=traffic_controls,
                           left_nbrs=left_nbrs,
                           right_nbrs=right_nbrs,
                           rotate_mat=rotate_mat,
                           size=size))
        return self.proj_drop(x_actor)

    def _ff_block(self, x_actor: torch.Tensor) -> torch.Tensor:
        return self.mlp(x_actor)
```

global_interactor_mod.py

```
from typing import Optional, List
import torch
import torch.nn as nn
from torch_geometric.nn.conv import MessagePassing
from torch_geometric.typing import Adj, OptTensor, Size
from torch_geometric.utils import softmax, subgraph
from utils import TemporalData, init_weights


class GlobalInteractorMod(nn.Module):

    def __init__(self,
                 historical_steps: int,
                 embed_dim: int,
                 edge_dim: int,
                 num_modes: int = 6,
                 num_heads: int = 8,
                 dropout: float = 0.1) -> None:
        super(GlobalInteractorMod, self).__init__()
        self.historical_steps = historical_steps
        self.embed_dim = embed_dim
```

```
        self.num_modes = num_modes

        self.global_interactor_layer = GlobalInteractorLayer(embed_dim=embed_dim,
                                                             edge_dim=edge_dim,
                                                             num_heads=num_heads,
                                                             dropout=dropout)
        self.multihead_proj = nn.Sequential(
            nn.LayerNorm(embed_dim),
            nn.Linear(embed_dim, num_modes * embed_dim))
        self.apply(init_weights)

    def forward(self, data: TemporalData, local_embed: torch.Tensor) -> torch.Tensor:
        edge_index, _ = subgraph(subset=~data['padding_mask'][:, self.historical_steps -
1], edge_index=data.edge_index)
        rel_pos = data['positions'][edge_index[0], self.historical_steps - 1] -
data['positions'][
            edge_index[1], self.historical_steps - 1]

        rel_pos = torch.bmm(rel_pos.unsqueeze(-2),
data['rotate_mat'][edge_index[1]]).squeeze(-2)
        rel_theta = data['rotate_angles'][edge_index[0]] -
data['rotate_angles'][edge_index[1]]

        x = self.global_interactor_layer(local_embed, rel_pos, rel_theta, edge_index)  #
[N, D]
        x = self.multihead_proj(x).view(-1, self.num_modes, self.embed_dim)  # [N, F, D]
        return x.transpose(0, 1)  # [F, N, D]


class GlobalNbrEmbed(nn.Module):
    '''Fuse Edge Attributes to Neighbor'''

    def __init__(self,
                 in_channels: List[int],
                 out_channel: int) -> None:
        super(GlobalNbrEmbed, self).__init__()
        self.agt_emb = nn.Sequential(
            nn.Linear(in_channels[0], out_channel, bias=False),
            nn.LayerNorm(out_channel),
            nn.ReLU(inplace=True),
            nn.Linear(out_channel, out_channel, bias=False))
        self.pos_emb = nn.Sequential(
            nn.Linear(in_channels[1], out_channel),
            nn.LayerNorm(out_channel),
            nn.ReLU(inplace=True),
            nn.Linear(out_channel, out_channel, bias=False))
        self.ang_emb = nn.Sequential(
            nn.Linear(in_channels[1], out_channel),
            nn.LayerNorm(out_channel),
```

```
            nn.ReLU(inplace=True),
            nn.Linear(out_channel, out_channel, bias=False))
        self.aggr_embed = nn.Sequential(
            nn.LayerNorm(out_channel),
            nn.ReLU(inplace=True),
            nn.Linear(out_channel, out_channel, bias=False),
            nn.LayerNorm(out_channel))
        self.apply(init_weights)

    def forward(self, continuous_inputs: List[torch.Tensor]) -> torch.Tensor:
        out = self.agt_emb(continuous_inputs[0]) + self.pos_emb(continuous_inputs[1]) +
self.ang_emb(continuous_inputs[2])
        return self.aggr_embed(out)


class GlobalInteractorLayer(MessagePassing):

    def __init__(self,
                 embed_dim: int,
                 edge_dim: int,
                 num_heads: int = 8,
                 dropout: float = 0.1,
                 **kwargs) -> None:
        super(GlobalInteractorLayer, self).__init__(aggr='add', node_dim=0, **kwargs)
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads
        self.scale = self.head_dim**0.5

        self.nbr_embed = GlobalNbrEmbed(in_channels=[embed_dim, edge_dim, edge_dim],
                                        out_channel=embed_dim)
        self.lin_q = nn.Linear(embed_dim, embed_dim, bias=False)
        self.lin_k = nn.Linear(embed_dim, embed_dim, bias=False)
        self.lin_v = nn.Linear(embed_dim, embed_dim, bias=False)
        self.lin_self = nn.Linear(embed_dim, embed_dim, bias=False)
        self.attn_drop = nn.Dropout(dropout)
        self.lin_ih = nn.Linear(embed_dim, embed_dim, bias=False)
        self.lin_hh = nn.Linear(embed_dim, embed_dim, bias=False)
        self.out_proj = nn.Linear(embed_dim, embed_dim)
        self.proj_drop = nn.Dropout(dropout)
        self.norm1 = nn.LayerNorm(embed_dim)
        self.norm2 = nn.LayerNorm(embed_dim)
        self.mlp = nn.Sequential(
            nn.Linear(embed_dim, embed_dim * 2),
            nn.ReLU(inplace=True),
            nn.Dropout(dropout),
            nn.Linear(embed_dim * 2, embed_dim),
            nn.Dropout(dropout))
```

```
    def forward(self,
                x: torch.Tensor,
                rel_pos: torch.Tensor,
                rel_theta: torch.Tensor,
                edge_index: Adj,
                size: Size = None) -> torch.Tensor:
        x = x + self._mha_block(self.norm1(x), rel_pos, rel_theta, edge_index, size)
        x = x + self._ff_block(self.norm2(x))
        return x

    def message(self,
                x_i: torch.Tensor,
                x_j: torch.Tensor,
                rel_pos: torch.Tensor,
                rel_theta: torch.Tensor,
                index: torch.Tensor,
                ptr: OptTensor,
                size_i: Optional[int]) -> torch.Tensor:
        x_j = self.nbr_embed([x_j,
                              rel_pos,
                              torch.cat((torch.cos(rel_theta).unsqueeze(-1),
                                         torch.sin(rel_theta).unsqueeze(-1)), dim=-1)])
        query = self.lin_q(x_i).view(-1, self.num_heads, self.head_dim)
        key = self.lin_k(x_j).view(-1, self.num_heads, self.head_dim)
        value = self.lin_v(x_j).view(-1, self.num_heads, self.head_dim)
        alpha = torch.sum((query * key), dim=-1, keepdim=True) / self.scale
        alpha = self.attn_drop(softmax(alpha, index, ptr, size_i))
        return value * alpha

    def update(self,
               inputs: torch.Tensor,
               x: torch.Tensor) -> torch.Tensor:
        inputs = inputs.view(-1, self.embed_dim)
        gate = torch.sigmoid(self.lin_ih(inputs) + self.lin_hh(x))
        return inputs + gate * (self.lin_self(x) - inputs)

    def _mha_block(self,
                   x: torch.Tensor,
                   rel_pos: torch.Tensor,
                   rel_theta: torch.Tensor,
                   edge_index: Adj,
                   size: Size) -> torch.Tensor:
        x = self.out_proj(self.propagate(edge_index=edge_index, x=x, size=size,
                                         rel_pos=rel_pos, rel_theta=rel_theta))
        return self.proj_drop(x)

    def _ff_block(self, x: torch.Tensor) -> torch.Tensor:
        return self.mlp(x)
```

decoder_mod.py

```
from typing import Tuple
import torch
import torch.nn as nn
from utils import init_weights
from models.layers import Linear, LinearRes


class DecoderMod(nn.Module):

    def __init__(self,
                 embed_dim: int,
                 future_steps: int,
                 num_modes: int,
                 uncertain: bool = True,
                 min_scale: float = 1e-3) -> None:
        super(DecoderMod, self).__init__()
        self.hidden_size = embed_dim
        self.future_steps = future_steps // 10
        self.num_modes = num_modes
        self.uncertain = uncertain
        self.min_scale = min_scale

        self.aggr_embed = nn.Sequential(
            nn.ReLU(inplace=True),
            LinearRes(self.hidden_size, self.hidden_size))
        self.loc = nn.Sequential(
            LinearRes(self.hidden_size, self.hidden_size),
            nn.Linear(self.hidden_size, self.future_steps * 2))
        self.scale = nn.Sequential(
            Linear(self.hidden_size, self.hidden_size, act=True),
            nn.Linear(self.hidden_size, self.future_steps * 2),
            nn.ReLU(inplace=True))
        self.pi = nn.Sequential(
            LinearRes(self.hidden_size, self.hidden_size),
            nn.Linear(self.hidden_size, 1))
        self.apply(init_weights)

    def forward(self, local_embed: torch.Tensor,
                global_embed: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        out = local_embed.unsqueeze(0) + global_embed
        out = self.aggr_embed(out)
        loc = self.loc(out).view(self.num_modes, -1, self.future_steps, 2)  # [F, N, H, 2]
        pi = self.pi(out).squeeze(-1).t()
        if self.uncertain:
            scale = self.scale(out).view(self.num_modes, -1, self.future_steps, 2) + 1.0
            return torch.cat((loc, scale), dim=-1), pi  # [F, N, H, 4], [N, F]
```

```
        else:
            return loc, pi  # [F, N, H, 2], [N, F]
```

layers.py

```
from torch import nn


class Linear(nn.Module):
    '''Linear Block: Linear -> Norm. -> ReLU (optional)'''

    def __init__(self, n_in, n_out, norm='LN', act=True, bias=False):
        super(Linear, self).__init__()
        assert (norm in ['LN', 'BN'])

        self.linear = nn.Linear(n_in, n_out, bias=bias)

        if norm == 'LN':
            self.norm = nn.LayerNorm(n_out)
        elif norm == 'BN':
            self.norm = nn.BatchNorm1d(n_out)

        self.relu = nn.ReLU(inplace=True)
        self.act = act

    def forward(self, x):
        out = self.linear(x)
        out = self.norm(out)
        if self.act:
            out = self.relu(out)
        return out


class LinearRes(nn.Module):
    '''Residual Linear Block: ResNetv1-like'''

    def __init__(self, n_in, n_out, norm='LN'):
        super(LinearRes, self).__init__()
        assert (norm in ['LN', 'BN'])

        self.linear1 = nn.Linear(n_in, n_out, bias=False)
        self.linear2 = nn.Linear(n_out, n_out, bias=False)
        self.relu = nn.ReLU(inplace=True)

        if norm == 'LN':
            self.norm1 = nn.LayerNorm(n_out)
            self.norm2 = nn.LayerNorm(n_out)
        elif norm == 'BN':
```

```
            self.norm1 = nn.BatchNorm1d(n_out)
            self.norm2 = nn.BatchNorm1d(n_out)

        if n_in != n_out:
            if norm == 'LN':
                self.transform = nn.Sequential(
                    nn.Linear(n_in, n_out, bias=False),
                    nn.LayerNorm(n_out))
            elif norm == 'BN':
                self.transform = nn.Sequential(
                    nn.Linear(n_in, n_out, bias=False),
                    nn.BatchNorm1d(n_out))
        else:
            self.transform = None

    def forward(self, x):
        out = self.linear1(x)
        out = self.norm1(out)
        out = self.relu(out)
        out = self.linear2(out)
        out = self.norm2(out)

        if self.transform is not None:
            out += self.transform(x)
        else:
            out += x

        out = self.relu(out)
        return out


class LinearRes2(nn.Module):
    '''Residual Linear Block: Inverted Bottleneck'''

    def __init__(self, n_in, n_out, norm='LN'):
        super(LinearRes2, self).__init__()
        assert (norm in ['LN', 'BN'])

        self.linear1 = nn.Linear(n_in, n_out * 2, bias=False)
        self.linear2 = nn.Linear(n_out * 2, n_out, bias=False)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        out = self.linear1(x[1])
        out = self.relu(out)
        out = self.linear2(out)
        return out + x[0]
```

train.py

```
from argparse import ArgumentParser

import numpy as np
import random
import torch
import torch.nn as nn
from utils import Logger, SpecsRecorder, Timer
import os

from datamodules import ArgoverseV1DataModule
from models.hivt import HiVT

from tqdm import tqdm
from datetime import datetime
from fvcore.nn import parameter_count_table

torch.multiprocessing.set_sharing_strategy('file_system')

parser = ArgumentParser()
parser.add_argument('--root', type=str, required=True)
parser.add_argument('--train_batch_size', type=int, default=32)
parser.add_argument('--val_batch_size', type=int, default=32)
parser.add_argument('--shuffle', type=bool, default=True)
parser.add_argument('--num_workers', type=int, default=8)
parser.add_argument('--pin_memory', type=bool, default=True)
parser.add_argument('--persistent_workers', type=bool, default=True)
parser.add_argument('--gpus', type=int, default=1)
parser.add_argument('--max_epochs', type=int, default=64)
parser.add_argument('--weight', type=str, default='')
parser.add_argument('--resume', action='store_true')
parser.add_argument('--name', type=str, default='')
parser.add_argument('--infer', action='store_true')
parser.add_argument('--per_sec', action='store_true')
parser.add_argument('--warmup_epochs', type=int, default=0)
parser.add_argument('--restart', action='store_true')
parser = HiVT.add_model_specific_args(parser)

file_path = os.path.abspath(__file__)
root_path = os.path.dirname(file_path)
# model_name = os.path.basename(file_path).split(".")[0]

seed = 42
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
np.random.seed(seed)
random.seed(seed)
```

```
def main():
    args = parser.parse_args()

    dataset = ArgoverseV1DataModule(args.root, args.train_batch_size, args.val_batch_size,
use_sampled_dataset=args.use_sampled_dataset)
    train_loader = dataset.train_dataloader()
    val_loader = dataset.val_dataloader()

    if torch.cuda.is_available():
        device_ = torch.device('cuda')
    else:
        device_ = torch.device('cpu')
    model = HiVT(**vars(args)).to(device_)

    decay = set()
    no_decay = set()
    whitelist_weight_modules = (nn.Linear, nn.Conv1d, nn.Conv2d, nn.Conv3d,
nn.MultiheadAttention, nn.LSTM, nn.GRU)
    blacklist_weight_modules = (nn.BatchNorm1d, nn.BatchNorm2d, nn.BatchNorm3d,
nn.LayerNorm, nn.Embedding)
    for module_name, module in model.named_modules():
        for param_name, param in module.named_parameters():
            full_param_name = '%s.%s' % (module_name, param_name) if module_name else
param_name
            if 'bias' in param_name:
                no_decay.add(full_param_name)
            elif 'weight' in param_name:
                if isinstance(module, whitelist_weight_modules):
                    decay.add(full_param_name)
                elif isinstance(module, blacklist_weight_modules):
                    no_decay.add(full_param_name)
            elif not ('weight' in param_name or 'bias' in param_name):
                no_decay.add(full_param_name)
    param_dict = {param_name: param for param_name, param in model.named_parameters()}
    inter_params = decay & no_decay
    union_params = decay | no_decay
    assert len(inter_params) == 0
    assert len(param_dict.keys() - union_params) == 0

    optim_groups = [
        {"params": [param_dict[param_name] for param_name in sorted(list(decay))],
            "weight_decay": model.weight_decay},
        {"params": [param_dict[param_name] for param_name in sorted(list(no_decay))],
            "weight_decay": 0.0}]

    optimizer = torch.optim.AdamW(optim_groups, lr=model.lr,
weight_decay=model.weight_decay)
    if args.warmup_epochs > 0:
```

```
        scheduler = torch.optim.lr_scheduler.LinearLR(optimizer=optimizer,
start_factor=0.2, end_factor=1.0, total_iters=args.warmup_epochs)
    else:
        if args.restart:
            scheduler =
torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer=optimizer, T_0=1, T_mult=2,
eta_min=0.0)
        else:
            scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer=optimizer,
T_max=args.max_epochs, eta_min=0.0)

    epoch = 0
    param_table = parameter_count_table(model)
    param_count = sum([param.nelement() for param in model.parameters()])
    if args.weight:
        state = torch.load(args.weight)
        model.load_state_dict(state['state_dict'])
        optimizer.load_state_dict(state['optimizer'])
        if args.resume:
            epoch = state['epoch']
            save_dir = os.path.dirname(args.weight)
            if epoch >= args.warmup_epochs:
                if args.restart:
                    scheduler =
torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer=optimizer, T_0=1, T_mult=2,
eta_min=0.0)
                else:
                    scheduler =
torch.optim.lr_scheduler.CosineAnnealingLR(optimizer=optimizer, T_max=args.max_epochs,
eta_min=0.0)
            else:
                scheduler = torch.optim.lr_scheduler.LinearLR(optimizer=optimizer,
start_factor=0.2, end_factor=1.0, total_iters=args.warmup_epochs)
        scheduler.load_state_dict(state['scheduler'])
        if 'random_state' in state.keys():
            torch.set_rng_state(state['random_state'])
            # np.random.set_state(state['random_state'])
    else:
        if args.name != "":
            save_dir = os.path.join(root_path, "results",
datetime.now().strftime("%Y-%m-%d"), args.name)
        else:
            save_dir = os.path.join(root_path, "results",
datetime.now().strftime("%Y-%m-%d"), datetime.now().strftime("%H-%M-%S"))
        if not os.path.exists(save_dir):
            os.makedirs(save_dir)
        if not os.path.exists(os.path.join(save_dir, "specs")):
            specs_recorder = SpecsRecorder(save_dir)
            specs_recorder.record(f"Number of params: {param_count:,}")
```

```
            specs_recorder.record(param_table)

    logger = Logger(save_dir)
    timer = Timer()

    while epoch < args.max_epochs + args.warmup_epochs:
        # Train
        timer.tic()
        model.train()
        for _, data in tqdm(enumerate(train_loader)):
            loss = model(data.to(device_))

            optimizer.zero_grad()
            loss["loss"].backward()
            optimizer.step()

        ade1, fde1, mr1, ade, fde, mr, b_fde, cls_loss_val, reg_loss_val, loss_val =
compute_metrics(model)

        epoch += 1
        logger.info(f"Epoch {int(epoch)}/{args.max_epochs + args.warmup_epochs}, lr
{scheduler.get_last_lr()[0]:.6f} "
                    f"————————————————————————————————————————————————————————————
————————————————————")
        logger.info(f"train: time {timer.toc():.2f}s, "
                    f"loss {loss_val:.4f}, cls {cls_loss_val:.4f}, reg
{reg_loss_val:.4f}\n"
                    f"k=1: minADE {ade1:.4f}, minFDE {fde1:.4f}, MR {mr1:.4f}, "
                    f"k=6: minADE {ade:.4f}, minFDE {fde:.4f}, MR {mr:.4f}, brier-minFDE
{b_fde:.4f}\n")

        scheduler.step()
        if epoch == args.warmup_epochs:
            if args.restart:
                scheduler =
torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer=optimizer, T_0=1, T_mult=2,
eta_min=0.0)
            else:
                scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer=optimizer,
T_max=args.max_epochs, eta_min=0.0)

        # Save
        state = {
            'epoch': epoch,
            'state_dict': model.state_dict(),
            'optimizer': optimizer.state_dict(),
            'scheduler': scheduler.state_dict(),
            'random_state': torch.get_rng_state()}
        torch.save(state, os.path.join(save_dir, str(epoch)) + '.pth')
```

```
        # Eval
        timer.tic()
        model.eval()
        for _, data in tqdm(enumerate(val_loader)):
            with torch.no_grad():
                model(data.to(device_))

        ade1, fde1, mr1, ade, fde, mr, b_fde, cls_loss_val, reg_loss_val, loss_val =
compute_metrics(model)

        logger.info(f"val: time {timer.toc():.2f}s, "
                    f"loss {loss_val:.4f}, cls {cls_loss_val:.4f}, reg
{reg_loss_val:.4f}\n"
                    f"k=1: minADE {ade1:.4f}, minFDE {fde1:.4f}, MR {mr1:.4f}, "
                    f"k=6: minADE {ade:.4f}, minFDE {fde:.4f}, MR {mr:.4f}, brier-minFDE
{b_fde:.4f}\n")

def compute_metrics(model):
    ade1 = model.minADE1.compute()
    fde1 = model.minFDE1.compute()
    mr1 = model.MR1.compute()
    ade = model.minADE.compute()
    fde = model.minFDE.compute()
    mr = model.MR.compute()
    b_fde = model.b_minFDE.compute()
    cls_loss_val = model.cls_loss_val.compute()
    reg_loss_val = model.reg_loss_val.compute()
    loss_val = model.loss_val.compute()

    model.minADE1.reset()
    model.minFDE1.reset()
    model.MR1.reset()
    model.minADE.reset()
    model.minFDE.reset()
    model.MR.reset()
    model.b_minFDE.reset()
    model.cls_loss_val.reset()
    model.reg_loss_val.reset()
    model.loss_val.reset()

    return ade1, fde1, mr1, ade, fde, mr, b_fde, cls_loss_val, reg_loss_val, loss_val


if __name__ == '__main__':
    main()
```

test.py

```
from argparse import ArgumentParser

import numpy as np
import random
import torch
from utils import Logger, SpecsRecorder
import os

from datamodules import ArgoverseV1DataModule
from models.hivt import HiVT

from tqdm import tqdm
import pandas as pd

torch.multiprocessing.set_sharing_strategy('file_system')

parser = ArgumentParser()
parser.add_argument('--root', type=str, required=True)
parser.add_argument('--val_batch_size', type=int, default=32)
parser.add_argument('--num_workers', type=int, default=8)
parser.add_argument('--pin_memory', type=bool, default=True)
parser.add_argument('--persistent_workers', type=bool, default=True)
parser.add_argument('--gpus', type=int, default=1)
parser.add_argument('--weight', type=str, default='')
parser.add_argument('--test', action='store_true')
parser.add_argument('--use_raw_data', action='store_true')
parser.add_argument('--per_sec', action='store_true')
parser.add_argument('--infer', action='store_true')
parser = HiVT.add_model_specific_args(parser)

file_path = os.path.abspath(__file__)
root_path = os.path.dirname(file_path)
# sys.path.insert(0, root_path)


def main():
    seed = 42
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    np.random.seed(seed)
    random.seed(seed)

    args = parser.parse_args()

    dataset = ArgoverseV1DataModule(args.root, args.val_batch_size, args.val_batch_size)
    if args.test:
        data_loader = dataset.test_dataloader()
```

```
    else:
        data_loader = dataset.val_dataloader()

    if torch.cuda.is_available():
        device_ = torch.device('cuda')
    else:
        device_ = torch.device('cpu')
    model = HiVT(**vars(args)).to(device_)

    state = torch.load(args.weight)
    model.load_state_dict(state['state_dict'])
    save_dir = os.path.join(os.path.dirname(args.weight), 'eval')
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)
    if not os.path.exists(os.path.join(save_dir, "specs")):
        from fvcore.nn import parameter_count_table
        param_table = parameter_count_table(model)
        param_count = sum([param.nelement() for param in model.parameters()])
        specs_recorder = SpecsRecorder(save_dir)
        specs_recorder.record(f"Number of params: {param_count:,}")
        specs_recorder.record(param_table)

    # begin inference
    preds, probs, cities, gts = {}, {}, {}, {}
    for _, data in tqdm(enumerate(data_loader)):
        with torch.no_grad():
            batch_argo_idx = data["seq_id"].numpy()
            pos, prob = model(data.to(device_))
            if not args.test:
                if not args.use_raw_data:
                    # local_gt = data['positions'][data['agent_index']][:, 20:]
                    gt = torch.bmm(data['positions'][data['agent_index']][:, 20:],
                                   data['global_rotate_mat']) + data['origin'].unsqueeze(1)
                    gt = {idx: each_gt for (idx, each_gt) in zip(batch_argo_idx, gt)}
                else:
                    gt = []
                    for argo_idx in batch_argo_idx:
                        raw_data = pd.read_csv(os.path.join(args.root,
f'val/data/{argo_idx}.csv'))
                        traj = raw_data[raw_data['OBJECT_TYPE'] == 'AGENT']
                        gt.append(np.stack([traj['X'].values, traj['Y'].values], axis=-
1)[20:])
                    gt = {idx: each_gt for (idx, each_gt) in zip(batch_argo_idx, gt)}

            for (argo_idx, pred_traj, pred_prob, city) in zip(batch_argo_idx, pos, prob,
data["city"]):
                preds[argo_idx] = pred_traj
                probs[argo_idx] = pred_prob
                cities[argo_idx] = city
```

```
            if not args.test:
                gts[argo_idx] = gt[argo_idx]
    torch.cuda.empty_cache()

    # evaluate or submit
    if not args.test:
        # save for visualization
        res = dict(preds=preds, probs=probs, gts=gts, cities=cities)
        np.save(f"{save_dir}/results--epoch{state['epoch']}.npy", res, allow_pickle=True)

        # for val set: compute metric
        from argoverse.evaluation.eval_forecasting import compute_forecasting_metrics
        metrics1 = compute_forecasting_metrics(preds, gts, cities, 1, 30, 2, probs)
        metrics6 = compute_forecasting_metrics(preds, gts, cities, 6, 30, 2, probs)
        logger = Logger(save_dir)
        logger.info(f"weight: {args.weight}\n"
                    f"------------------------------------------------\n"
                    f"Prediction Horizon : 30, Max #guesses (K): 1\n"
                    f"------------------------------------------------\n"
                    f"{metrics1}\n"
                    f"------------------------------------------------\n"
                    f"------------------------------------------------\n"
                    f"Prediction Horizon : 30, Max #guesses (K): 6\n"
                    f"------------------------------------------------\n"
                    f"{metrics6}\n"
                    f"------------------------------------------------\n", verbose=False)
    else:
        # for test set: save as h5 for submission in evaluation server
        from argoverse.evaluation.competition_util import generate_forecasting_h5
        model_name = os.path.basename(os.path.dirname(args.weight))
        generate_forecasting_h5(data=preds,
                                output_path=f"{save_dir}/submission",
                                filename=f"{model_name}--epoch{state['epoch']}",
                                probabilities=probs)


if __name__ == '__main__':
    main()
```