VII. Supplementary

A. Details of the U-Net inspired attention mechanism

Recall that in Section III of the main paper, we had used joint trainable parameters of a U-Net inspired architecture with shared encoder weights for determining $F_{value}, F_{query}, F_{key}$. In this subsection we elaborate more on this architecture and its implementation.

The idea of U-Attention originates from first integrating together the characteristics of TransformerConv[13] and EdgeConv[14]. TransformerConv applies attention on the neighbouring nodes but does not concatenate information of the ego-node. In contrast, EdgeConv concatenates relative information of the neighbouring nodes with that of the ego-node. However, information aggregation is done without deciding which node to pay more attention to. Our first step is to first combine these two ideas, i.e.:

$$F_{Att}^{l+1}(z_i^l, z_j^l) = W_{self}^l \cdot z_i^l + \sum_{j \in N_i} \alpha_{ij} \cdot W_{value}(z_i^l | z_i^l - z_j^l) \quad (5)$$

where $\alpha_{ij} = \frac{W_{query}(z_i^l)^T \cdot W_{key}(z_j^l)}{\sum_{k \in N_i} W_{query}(z_i^l)^T \cdot W_{key}(z_k^l)}$ and $W_{value}, W_{query}, W_{key}$ are the trainable parameter matrices. For simplicity, we set $x_j^l = (z_i^l | z_i^l - z_j^l)$. If we only consider operations within a layer, we can eliminate the need for the layer index l, so x_j^l can be further simplified to x_j. The key, query and value of x_j is respectively denoted by K_j, Q_j and V_j. In the traditional transformer block, K_j, Q_j and V_j can be calculated by $K_j = W_{key} \cdot x_j$, $Q_j = W_{query} \cdot x_j$ and $V_j = W_{value} \cdot x_j$.

Next step, we look at the formulation of self-attention weights α_{ij}:

$$\begin{aligned} \alpha_{ij} &= \frac{K_j^T \cdot Q_j}{\sum_{k \in N_i} K_k^T \cdot Q_j} \\ &= \frac{(W_{key} \cdot x_j)^T \cdot (W_{query} \cdot x_j)}{\sum_{k \in N_i} (W_{key} \cdot x_k)^T \cdot (W_{query} \cdot x_j)} \\ &= \frac{x_j^T \cdot (W_{key}^T \cdot W_{query}) \cdot x_j}{\sum_{k \in N_i} x_k^T \cdot (W_{key}^T \cdot W_{query}) \cdot x_j} \end{aligned} \quad (6)$$

Note that we can treat the matrix multiplication $(W_{key}^T \cdot W_{query})$ in the above equation as a single entity, in which the output has the same dimension as the input just as in the case of an autoencoder. Here we can treat W_{key} as the encoder while W_{query} as the decoder. Both the encoder and the decoder have only one single linear layer without any non-linearity. Inspired by this interpretation, we use the auto-encoder to directly calculate attention α. Skip connections are added between the encoder and decoder for stability. With these changes we have the neural networks F_{query}, F_{key} instead of the original matrices W_{key} and W_{query} and Equation 6 is modified to:

$$\begin{aligned} \alpha_{ij} &= \frac{F_{query}(F_{key}(x_j))^T \cdot x_j}{\sum_{k \in N_i} F_{query}(F_{key}(x_k))^T \cdot x_j} \\ &= \frac{F_{query\_key}(x_j))^T \cdot x_j}{\sum_{k \in N_i} F_{query\_key}(x_k))^T \cdot x_j} \end{aligned} \quad (7)$$

Meanwhile, the second term of Equation 5 can be re-written as:

$$\begin{aligned} \sum_{j \in N_i} \alpha_{ij} \cdot W_{value}(z_i | z_i - z_j) &= \sum_{j \in N_i} \alpha_{ij} \cdot W_{value} \cdot x_j \\ &= \sum_{j \in N_i} \alpha_{ij} \cdot V_j \end{aligned} \quad (8)$$

where α_{ij} is calculated by a U-net attention network described earlier. It is easy to notice that this U-net has a latent vector $F_{key}(x_j)$ at the output of the encoder, which already encodes information about x_j. This information can be used to determine the value V as it is also a function of x_j. Originally, the value V is given by a simple matrix multiplication between the input x_j and the matrix W_{value}. Instead of this we pass the the latent vector $F_{key}(x)$ through an independent decoder F_{value} to determine the value of V Skip connections are additionally added from encoder F_{key} to this decoder head F_{value}. So the value can be calculated by:

$$\begin{aligned} V_i &= F_{value}(F_{key}(x_j)) \\ &= F_{value\_key}(x_j) \end{aligned} \quad (9)$$

Hence, by replacing the linear layers W_{key}, W_{query} and W_{value} in Equation 5 with Equation 7 and Equation 9, we will get the final equation of our model Equation 1. Figure 4 shows the schematic of our U-Attention Block.

B. Ablation studies on the contribution of the different components of the model

In Subsection VII-A, we described the two important adjustments to our architecture which differentiates it from TransformerConv [13]. First, we concatenate the relative information of neighbouring node with the information of self node. Next, we replaced the normal transformer block with the U-Attention block. In this subsection, we investigate the contribution of each of these adjustments on the overall performance. Therefore, we conduct an additional ablation study where our model is compared against one that removes the U-Attention structure and another that does not concatenate information from the self-node for determining attention weights. Eliminating both adjustments collapses our model to being TransformerConv. The training was done on the same dataset as described in the main paper. Table II shows the performance of these model variants on different vehicle/obstacle combinations.

From Table II, we notice that, adjusting the attention block to remove the U-Net architecture but keeping the concatenation leads to a drop in performance. This demonstrates the importance of using the U-Net architecture, without which performance deteriorates. The reason is that the traditional TransformerConv architecture only uses a simple

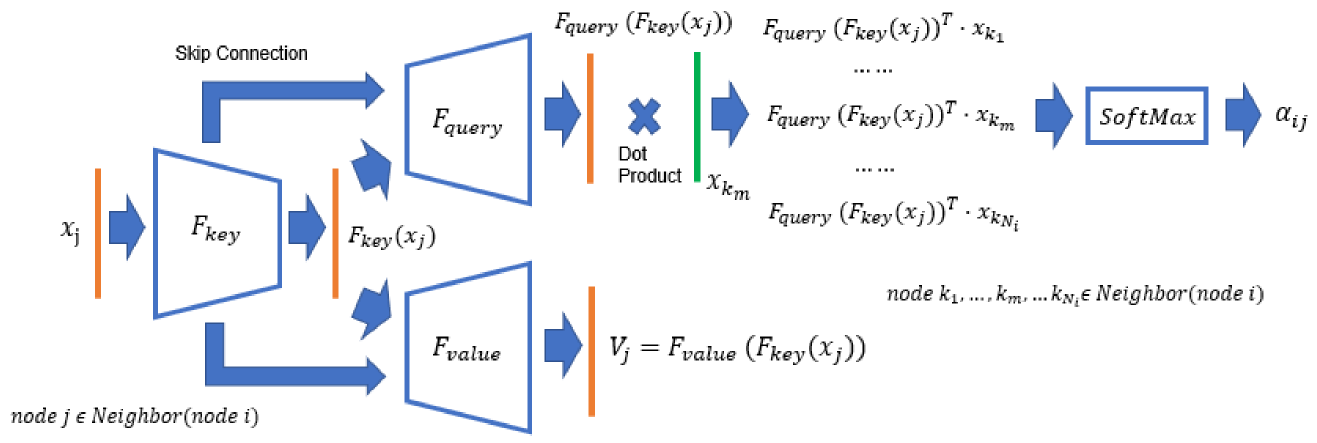
Skip Connection
x_j
F_{key}
$F_{key}(x_j)$
F_{query}
F_{value}
$F_{query}(F_{key}(x_j))$
Dot Product
x_{k_m}
$F_{query}(F_{key}(x_j))^T \cdot x_{k_1}$
……
$F_{query}(F_{key}(x_j))^T \cdot x_{k_m}$
……
$F_{query}(F_{key}(x_j))^T \cdot x_{k_{N_i}}$
$SoftMax$
α_{ij}
$node\ k_1, \ldots, k_m, \ldots k_{N_i} \epsilon\ Neighbor(node\ i)$
$V_j = F_{value}(F_{key}(x_j))$
$node\ j\ \epsilon\ Neighbor(node\ i)$

Fig. 4: Schematics of the U-Attention Block

		success to goal rate				collision rate			
Num. Veh.	Num. Obs.	Our Model	Remove Concatenation	Remove Unet	TransformerConv	Our Model	Remove Concatenation	Remove Unet	TransformerConv
1	0	1.0000	0.9847	0.9990	0.9894	-	-	-	-
1	1	0.9677	0.9161	0.7307	0.7580	4.9616E-05	8.9603E-04	5.1616E-03	5.3226E-03
1	2	0.8991	0.8109	0.5583	0.5721	3.3231E-04	2.2233E-03	1.1090E-02	1.1451E-02
1	3	0.8127	0.7093	0.4677	0.4929	6.6306E-04	3.9955E-03	1.5917E-02	1.6024E-02
1	4	0.7361	0.6302	0.3764	0.3922	1.2774E-03	5.4084E-03	2.1584E-02	2.1938E-02
2	0	0.9984	0.9702	0.6472	0.6600	1.1006E-05	6.0761E-04	8.0570E-03	7.6941E-03
2	1	0.9634	0.8688	0.5940	0.6219	1.6017E-04	3.0018E-03	1.0235E-02	9.6096E-03
2	2	0.8676	0.7810	0.5425	0.5624	4.5357E-04	4.6598E-03	1.3381E-02	1.2796E-02
2	3	0.7792	0.6955	0.5142	0.5209	6.7795E-04	6.4930E-03	1.6038E-02	1.6039E-02
2	4	0.6781	0.6135	0.4899	0.4982	1.1135E-03	8.2745E-03	1.9067E-02	1.8711E-02
3	0	0.9943	0.8107	0.4129	0.4339	8.8107E-05	3.7200E-03	1.4707E-02	1.4047E-02
3*	1*	0.9706	0.7489	0.3950	0.4193	3.0382E-04	5.2493E-03	1.6100E-02	1.5655E-02
3*	2*	0.9302	0.7006	0.3897	0.4174	5.9881E-04	6.6222E-03	1.7780E-02	1.7393E-02
3*	3*	0.8903	0.6503	0.3754	0.4023	8.9677E-04	7.9807E-03	2.0166E-02	1.9646E-02
3*	4*	0.8328	0.6161	0.3554	0.3834	1.6090E-03	9.2047E-03	2.2478E-02	2.2045E-02
4*	0*	0.9807	0.6607	0.2894	0.2895	2.7650E-04	6.5573E-03	2.0186E-02	1.9967E-02
4*	1*	0.9550	0.6185	0.2701	0.3048	5.7179E-04	7.8322E-03	2.1356E-02	2.0446E-02
4*	2*	0.9279	0.5804	0.2773	0.2905	9.4375E-04	8.7571E-03	2.1877E-02	2.1960E-02
4*	3*	0.8853	0.5426	0.2773	0.2864	1.4612E-03	1.0007E-02	2.3559E-02	2.3747E-02
5*	0*	0.9590	0.5322	0.1890	0.1973	5.8217E-04	9.1856E-03	2.6257E-02	2.5964E-02
5*	1*	0.9285	0.4856	0.1934	0.2078	1.0483E-03	1.0374E-02	2.6303E-02	2.6115E-02
5*	2*	0.9037	0.4759	0.2078	0.2125	1.4063E-03	1.0754E-02	2.6261E-02	2.6607E-02
6*	0*	0.9209	0.4128	0.1300	0.1347	1.1376E-03	1.1734E-02	3.1954E-02	3.1612E-02
6*	1*	0.8949	0.3916	0.1419	0.1556	1.5096E-03	1.2351E-02	3.0973E-02	3.0917E-02
6*	2*	0.8717	0.3738	0.1423	0.1479	1.8932E-03	1.2905E-02	3.1218E-02	3.1828E-02

TABLE II: An ablation study showing the performance of the four variants of our model: Full Version of Our Model, Our Model without concatenation, Our Model with the Unet structure removed and the original version of TransformerConv [13]). The metrics used for benchmarking are *success-to-goal rate* and *collision rate*. The evaluation is done on a completely unseen test data comprising of scenarios with 1-6 vehicles and 0-4 obstacles as shown by the corresponding rows. Each row in the table is evaluated on 4062 scenarios. The rows labeled with an asterisk (*) are those vehicle-obstacle combinations that were not even in the training set. The training set only comprised of 1-3 and 0-4 obstacles.

linear transform $W_{key}^T \cdot W_{query}$ to align the key and query. This is apparently not powerful enough to extract sufficient information when calculating attention. By adjusting the attention block to a U-Net architecture, the network in the attention block is much more powerful. So it is capable of extracting more rich and representative features which facilitates computing a more accurate attention value.

Similar observation holds for when using only the Unet architecture but removing the concatenation. The performance again deteriorates demonstrating the signinfcance of using concatenation. True performance gains are only realized when both the concatenation and Unet architecture are used together in conjunction.

C. Run Time Comparison

An advantage of our GNN model against traditional optimization based techniques is faster inference. Moreover, as the number of vehicles/obstacles is increased, the time taken by an optimization based procedure to find the correct control commands rises accordingly. However, with our graphical based architecture which allows parallel computations, the inference time remains fairly consistent. To demonstrate this, we run both the optimization method and our GNN model with increasing number vehicles and obstacles. For each vehicle/obstacle combination, we choose 100 cases from test dataset.

Note that an increase in the distance between start and destination state as well as the position of the static obstacles

Num. Veh.	Num. Obs.	Optimization	Our model (GPU)	Our model (CPU)
1	0	0.63230	0.00583	0.00408
1	1	0.73012	0.00781	0.00492
1	2	0.74968	0.00791	0.00522
1	3	0.75063	0.00800	0.00541
1	4	0.79725	0.00792	0.00568
2	0	2.74806	0.00783	0.00515
2	1	3.10209	0.00797	0.00570
2	2	3.06539	0.00797	0.00590
2	3	2.96316	0.00799	0.00634
2	4	2.96338	0.00799	0.00673
3	0	5.48114	0.00795	0.00591
3*	1*	-	0.00804	0.00686
3*	2*	-	0.00801	0.00607
3*	3*	-	0.00799	0.00609
3*	4*	-	0.00818	0.00635
4*	0*	-	0.00798	0.00577
4*	1*	-	0.00854	0.00622
4*	2*	-	0.00864	0.00657
4*	3*	-	0.00862	0.00679
5*	0*	-	0.00857	0.00644
5*	1*	-	0.00830	0.00692
5*	2*	-	0.00886	0.00741
6*	0*	-	0.00886	0.00732
6*	1*	-	0.00902	0.00780
6*	2*	-	0.00883	0.00808

TABLE III: Shows the average run time per step for the optimization based procedure and our GNN model. Note that our model takes less than 10 milliseconds to get a prediction for a scenario with 6 vehicles and 2 obstacles. Meanwhile, the optimization takes more than half second (632 milliseconds) even for the simplest case with 1 vehicle and no obstacle.

will influence the length of the trajectory and thereby increase the prediction run time. Therefore, rather than reporting the time to complete the entire trajectory, we report the average time to complete one step towards the destination in the trajectory.

Keeping in line with the main paper, we use the same parameters for online inference here as we did for offline optimization for the purpose of label generation. The resource we use for computation is an Intel Core i7-10750H. Table III reports the run time for both the optimization based method and our model. Our model involves multiple parallel computations and can therefore be readily be deployed on a GPU too. Runtime performance of our our model on a GeForce RTX 2070 GPU are also reported in the table.

From the results in Table III, it is evident that our model takes less than 10 milliseconds to get a prediction even for a scenario with 6 vehicles and 2 obstacles. Meanwhile, the optimization takes more than half second (632 milliseconds) even for the simplest case with 1 vehicle and no obstacle. There are two main reasons why the optimization runs much slower than our model. Firstly, the optimization procedure needs to be completed in real-time during inference, while our GNN model has already optimized its parameters during the training process. Although it takes longer to train the GNN model but once the parameters of the GNN are fixed, it is only a matter of using these pre-trained weights at inference time. But for optimization, each inference is a new problem, which needs to be iteratively solved at each step and thereby accumulating the total time consumed. Besides, optimization uses a receding horizon strategy. This means in order to execute one step, we still need to predict and calculate many steps ahead, which is not an effective use of computation. But these calculations are necessary, otherwise the optimization would not be able to look ahead in order to take pre-emptive action to avoid collision in advance. One can reduce the prediction horizon, but it may lead to a sub-optimal trajectory.

Another important point worth noticing, is that when the task gets more complex from 1 vehicle and 0 obstacle to 6 vehicles and 2 obstacles, the prediction run time of our GNN model only goes up marginally. But for optimization, the prediction run time increase dramatically when the number of vehicles increases. This is expected, because more vehicles means more constraints to be optimized for. Lastly, note that our model runs faster on the CPU than on the GPU. That is because during inference, the batch size is 1. Therefore, the advantage of parallelism gained from a forward pass on one sample does not compensate for the time it takes to transfer data between the CPU and GPU memory. Nevertheless, the model is still faster than compared with the optimization procedure. Moreover, GPU's are still advantageous during training wherein large batch sizes can be processed.

D. Breakdown of the training data

Note that Subsection IV-A mentioned that the training data for which labels are generated contain between 1-3 vehicles and 0-4 static obstacles, for a total of around 20,961 trajectories. The start and destination states of the vehicles/obstacles are generated at random. Each trajectory is collected for 120 timesteps. Therefore, the total number of scenarios generated are 2,515,320. A breakdown of this number for the different number of vehicles (1-3) and static obstacles (0-4) is shown in Table IV. As we saw in Subsection VII-C, increasing the number of vehicles/obstacles beyond what is enumerated in Table IV, significantly slows down the optimization for determining the optimal control values during this data and label generation process. Nevertheless, we demonstrated in Table III that our model is still powerful enough to make fast inference for up to 6 vehicles. This is despite being trained with data that had a maximum of 3 vehicles.

Num. Veh.	Num. Obs.	Num. Traj.	Num. Case
1	0	1000	120000
1	1	1200	144000
1	2	1800	216000
1	3	2699	323880
1	4	3289	394680
2	0	2000	240000
2	1	600	72000
2	2	1199	143880
2	3	1794	215280
2	4	2380	285600
3	0	3000	360000

TABLE IV: Amount of Ground Truth Data collected for 1-3 vehicles and 0-4 obstacles

When running the optimization, we set the prediction horizon to be 20. The steering angle is bounded within a range of -0.8 to 0.8 radians while the pedal acceleration is bounded between -1 to 1. Additional data is collected to make the model more robust to deviations at inference time. We simulate the vehicles diverging to random offset locations by adding noise on the position and orientation of the vehicle. After executing the first control command optimized for the the entire horizon, we add a random Gaussian noise on the position and orientation of the vehicle. The variance of the Gaussian noise decreases over time as the vehicle approaches the target position. This is to ensure that the vehicle can eventually reach its destination with minimal deviation.

E. *Conservative optimization behaviour*

Note that the in Table I of the main paper, the success-to-goal rate for our model is not a perfect score of 1. The reason the model fails to always reach the target destination is that it tends to behave conservatively. Rather than taking the risk of collision, it sometimes stops mid-way before other objects to avoid collision. This is because the ground truth data obtained from the optimization is not always prefect. Figure 5 shows a ground truth trajectory wherein the optimization failed to produce the requisite control commands to reach the target. Rather, the vehicle gets stuck midway. Nevertheless, it is important to keep such failed samples in the dataset for training as they teach the model to learn to stop before other objects to avoid collision. If these failure samples are removed from the training set, then the model performs worse as it starts colliding with other obstacles.

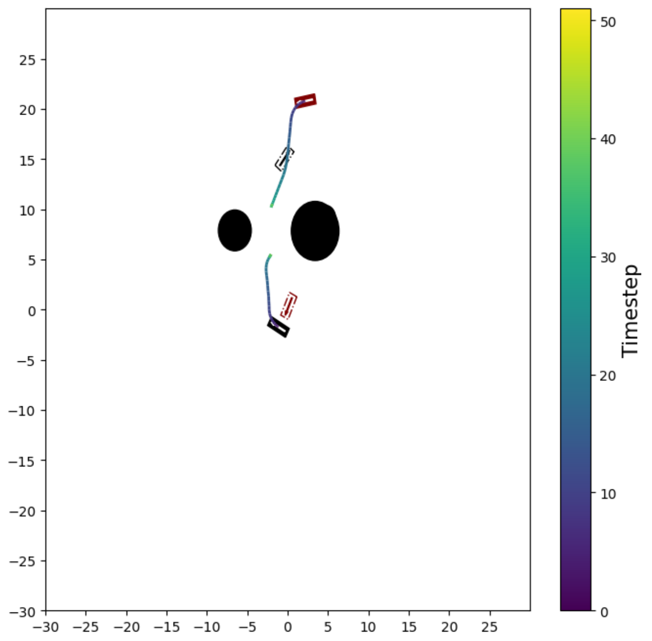

Fig. 5: An example of a conservative optimization yielding a sample trajectory wherein the vehicles halt rather than take the risk of collision in an attempt to reach their destination.