A Implementation details

We implement our model using Pytorch[2]. Here, we provide details of our model architecture, ablations and training.

A.1 Map representation

The nuScenes map API provides lane polylines, their successors, and polygons for cross-walks and stop lines. We consider map elements within an area of [-50, 50] m laterally and [-20, 80] m longitudinally around the target vehicle. This ensures that most ground truth trajectories lie within the area of interest. We split longer lane centerlines into snippets of maximum length 20m, and discretize the polylines at a 1m resolution. Each snippet corresponds to a node in the graph. This ensures that each lane node represents a lane segment of similar length. The node resolution (20m) and pose resolution (1m) for the polylines were experimentally chosen. There is a trade-off associated with the resolution of lane nodes: A finer resolution would provide a more informative set of inputs, but would lead to a graph with a greater number of nodes (and a greater number of poses per node) increasing encoder complexity.

A.2 GRU encoders

We embed both agent and node features using linear layers of size 16, followed by a leaky ReLU non-linearity. We use GRUs with depth 1 and hidden state dimension 32 on top of the embeddings for both the agent and node encoders.

A.3 Agent-node attention

We use scaled dot-product attention with a single attention head for the agent-node attention layers. We use 32×32 weight matrices for projecting the node and agent encodings for obtaining the queries, and keys and values respectively. The outputs of the attention layer are concatenated with the original node encodings and passed through a linear layer of size 32, followed by a leaky ReLU non-linearity to obtain updated node encodings of the same size as the original node encodings.

A.4 GNN layers

We use Pytorch geometric[3] for implementing the GCN and GAT layers of our model. For GCN layers, we use the layer-wise propagation rule from [5]. Our adjacency matrix includes both successor and proximal edges (treated as bidirectional), as well as self loops. The outputs at each node have the same dimension, 32, as the inputs. For GAT layers, we use the layer-wise propagation rule from [25]. We use a single attention head, with the outputs again having the same dimension as the inputs.

A.5 Policy header

The policy header is implemented as an MLP with 2 hidden layers of size 32 each and a scalar output. The input to the policy header for each edge is a vector of size 98, consisting of the source node encoding, destination node encoding and motion encoding of the target agent each of size 32, and a one-hot encoding for the edge type of size 2.

A.6 Trajectory decoder

We aggregate context along nodes traversed by the policy using a multi-head scaled dot-product attention layer. The attention layer has 32 parallel attention heads, and outputs a context vector $\mathcal{C}$ of size 128. We model the latent variable as a multivariate standard normal distribution. $z \sim \mathcal{N}(0, \mathbf{I})$, where $\mathbf{I}$ is a 5×5 identity matrix. We output a trajectory for each sampled $\mathcal{C}_k$, z_k and h_{motion} using an MLP with a hidden layer of size 128, and output of size 24 (x and y co-ordinates over the prediction horizon of 6 seconds at 2 Hz). We sample 200 trajectories from the model and cluster to obtain K=10 trajectories during training to compute the winner takes all regression loss $\mathcal{L}_{reg}$.

[2] https://pytorch.org/

[3] https://github.com/rusty1s/pytorch_geometric

A.7 Training

We train the model using Adam, with learning rate 1e-4, and a batch size of 32. For the first few epochs of training, since π_{route} does not produce meaningful traversals, we use the ground truth traversal for sampling trajectories and computing $\mathcal{L}_{reg}$. We pre-train the model using the ground truth traversal for 100 epochs. We then finetune using paths sampled from π_{route} for 100 epochs. We train our model using an AWS "p3-8xlarge" instance with 4 NVIDIA Tesla V100 GPUs. Each pre-training epoch takes roughly 1 minute and each finetuning epoch takes roughly 5 minutes for nuScenes.

A.8 Ranking Clustered Trajectories

The nuScenes leaderboard[4] requires a single set of ranked or scored predictions for computing the MinADE_k and MissRate metrics for k = 1, 5 and 10. We rank our set of 10 clustered trajectories based on Ward's merging cost[5]. We obtain the two clusters with the minimum merging cost. The trajectory corresponding to the smaller of the two clusters is assigned rank 10. The two clusters are then merged, with the merged cluster assuming the identity of the larger cluster. This process is then repeated to assign ranks 9 through 1. Using Ward's merging cost ensures that the top k trajectories cover a diverse set of modes for all values of k.

A.9 Decoder ablation details

MTP: For the MTP header, we first aggregate context over the entire graph using a multi-head scaled dot-product attention layer identical to our trajectory decoder, with 32 parallel attention heads and an output context vector $\mathcal{C}$ of size 128. We then use two fully connected layers of size 240 and 10 respectively to output K=10 trajectories, and K probabilities.

LV only: For the LV only decoder, similar to the MTP header, we first aggregate context over the entire graph using a multi-head attention layer with 32 attention heads and output $\mathcal{C}$ of size 128. The decoder then outputs trajectories conditioned on $\mathcal{C}$, h_{motion} and a sample z_k of the latent variable using the final MLP layer.

Traversal only: The traversal only decoder is identical to the trajectory decoder of our complete model, except for the final MLP layer, which outputs trajectories conditioned only on $\mathcal{C}_k$ and h_{motion} and not on the sampled latent variable z_k.

Goals + LV: The Goals + LV decoder consists of two output headers: A goal prediction header that outputs a scalar score at each node normalized using a softmax layer to give goal probabilities, and a trajectory decoder that outputs goal conditioned trajectories. We model the goal prediction header using an MLP with 2 hidden layers, each of size 32, and a scalar output. The input to the goal prediction header at each node is obtained by concatenating h_{node} and h_{motion}. The trajectory decoder consists of a multi-head attention layer with 32 heads that aggregates context over the entire graph to output a context vector $\mathcal{C}$ of size 128. $\mathcal{C}$ is concatenated with h_{motion}, a sampled latent vector z_k and the node encoding of a sampled goal $h_{node}^{u_k}$ and passed through an MLP with a hidden layer of size 128, and output size 24 corresponding to a goal conditioned trajectory.

[4] https://eval.ai/web/challenges/challenge-page/591/leaderboard/1659

[5] https://en.wikipedia.org/wiki/Ward%27s_method