TRISTAN - TRI SiTuation and Trajectory Anticipation Networks

Design/Technical Overview

Table of Contents

Goals **4**

Trajectory Predictor **4**

Architecture 4

Predictor Overview (Generator) 4

Full GAN Structure (discriminator is at training time) 5

Map Encoding 6

Input Encoders 6

Considerations for Online Use 7

Training Loss / Costs Structure 7

Generator Costs 7

Discriminator Costs 8

Coding/Design Guidelines **8**

General Rules 8

Testing Rules 9

Available In-Memory Data Items in the Data Loader **9**

Module Documentation **10**

Training Script 10

Pre-training / Curriculum Learning 10

Evaluation Runner 10

Metrics Computation 10

Criteria 1 (Miss Rate) 11

Criteria 2 (Predict Pedestrian Crossing) 11

Data Harvesting / Dataset Creation 11

Cloud Infrastructure 11

Tlog Snipping 11

Tlog Harvesting 11

Reharvesting 12

Pytorch Data Loader / Dataset Class 12

Decoder 12

Latent Factors 12

Trainer 13

Caching 13

Logging Handlers 14

SaveErrorStatistics 14

WorstCaseLoggerHandler 14

Visualization 14
Ablation Tests 15
Parameter sweeps 15

Available Tensorboard Statistics **16**
Scalar Categories 17
Image Entries/Tab 19
Text Entries/Tab 20
Runner Statistics Script 20

Keyword Material **20**
Map and Coordinates Handling 20
Model and Training Overview 21

This repository is the work of several people in Toyota Research Institute and collaborators. See the LICENSE.md for more information. If you use it for research purpose, please cite:

```
@inproceedings{huang2022:hyper,
  title={HYPER: Learned Hybrid Trajectory Prediction via Factored Inference and Adaptive
Sampling},
  author={Huang, Xin and Rosman, Guy and Gilitschenski, Igor and Jasour, Ashkan and McGill,
Stephen G. and Leonard, John J. and Williams, Brian C.},
  booktitle={2022 IEEE International Conference on Robotics and Automation (ICRA)},
  year={2022}
}

@inproceedings{kuo2022:language-traj,
  title={Trajectory Prediction with Linguistic Representations},
  author={Kuo, Yen-Ling and Huang, Xin and Barbu, Andrei and McGill, Stephen G. and Katz, Boris
and Leonard, John J. and Rosman, Guy},
  booktitle={2022 IEEE International Conference on Robotics and Automation (ICRA)},
  year={2022}
}
```

Goals

This document outlines the design of a software package that is implementing trajectory predictors and providing a framework to simplify trajectory prediction research at TRI/RAD. The package consists of:

1. Protocol Buffer interface for feeding trajectory data into a prediction neural network.
2. Interfaces and pipeline for building/training trajectory predictors
3. Helper functions, classes.

Specifically, the package makes it easy to experiment with different losses, predictor internal structures, and input processing networks.

Trajectory Predictor

Architecture

- The overall approach is an encoder-decoder. We have a GAN and non-GAN versions.
- Predictor/Generator:
    - The encoder uses a multi-agent spatiotemporal graph neural network that encodes agents' positions and other inputs (such as bounding box images, scene images).
    - The decoder is multiagent, LSTM-based, with different possible backbones ("latent factors").
    - The predictor emits samples of trajectory sets for all agents.
    - Can emit labels in parallel ("intent to cross").
- Discriminator:
    - Composed of two multiagent encoders -- one for the past, one for the future trajectory sets (used only in training).

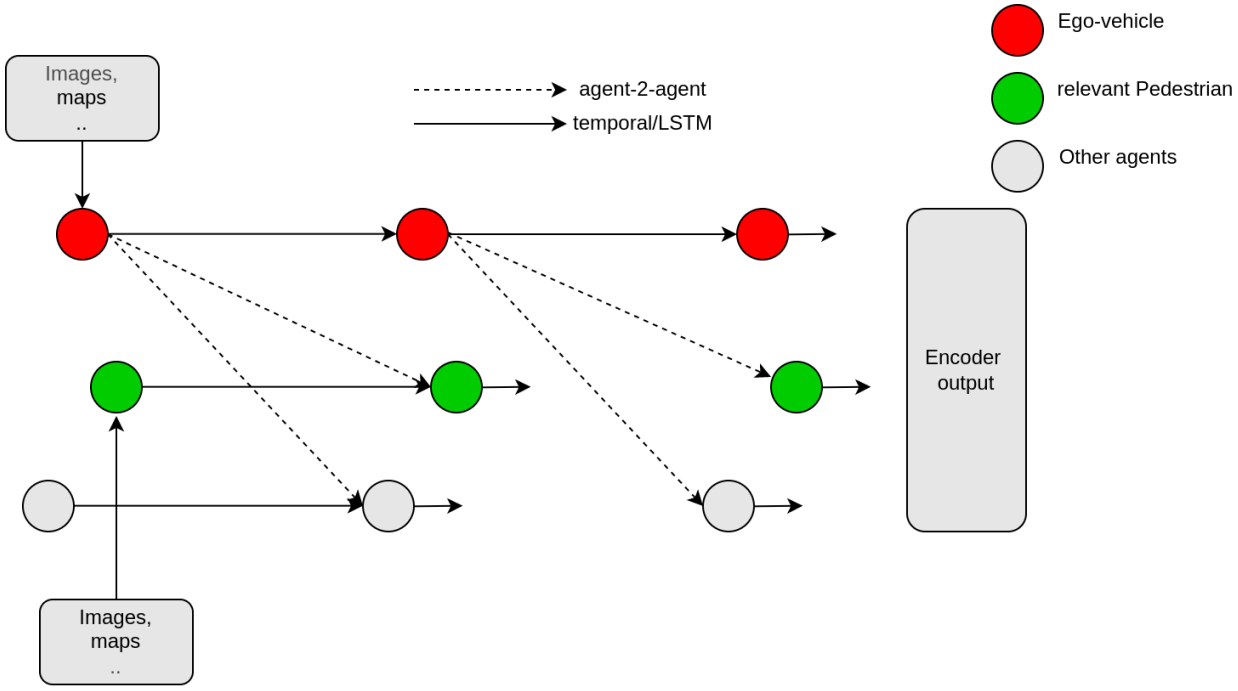
Additional important files:

- `./model_zoo/intent/prediction_model_interface.py` - encapsulates trajectory encoding-decoding along with encoding of different inputs, and generation of auxiliary outputs.

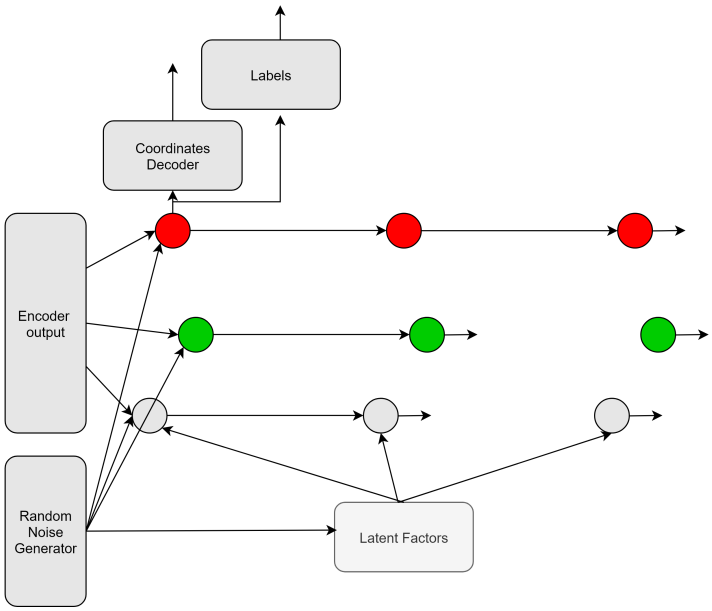
- Trainer - `./intent/multiagents/prediction_trainer.py` contains the main trainer class. See “Trainer” section below.
- Encoders
    - Graph_encoder - `./model_zoo/intent/batch_graph_encoder.py` contains a spatio-temporal graph encoder that processes the past inputs for the different agents.
    - Coordinate, Map encoders - see “Input Encoders” below.
- Running script - “Training Script” below.
- Latent factors interface - see “Latent Factors” below. `./model_zoo/intent/prediction_latent_factors.py`, captures internal representation to be used during decoding, for capturing different inductive biases.

Predictor Overview (Generator)

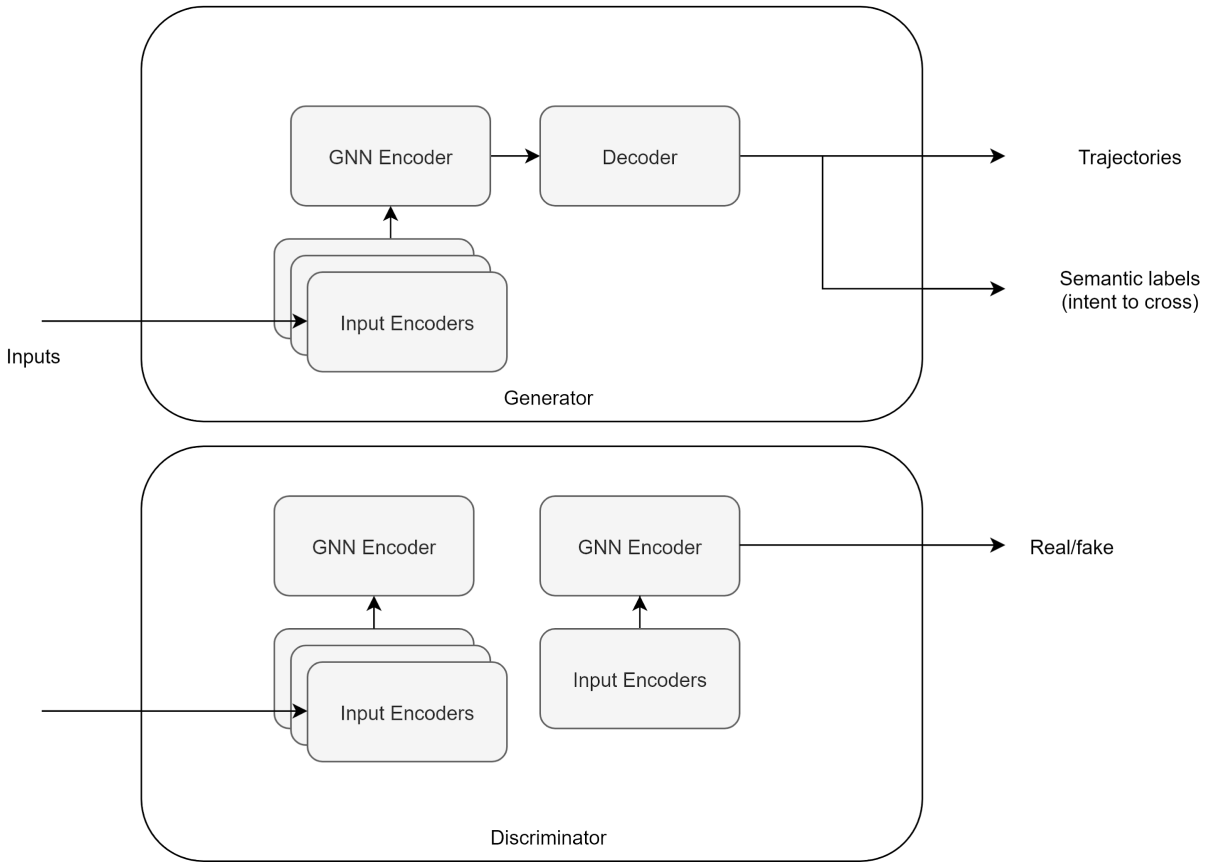
Encoder block:



Decoder block:

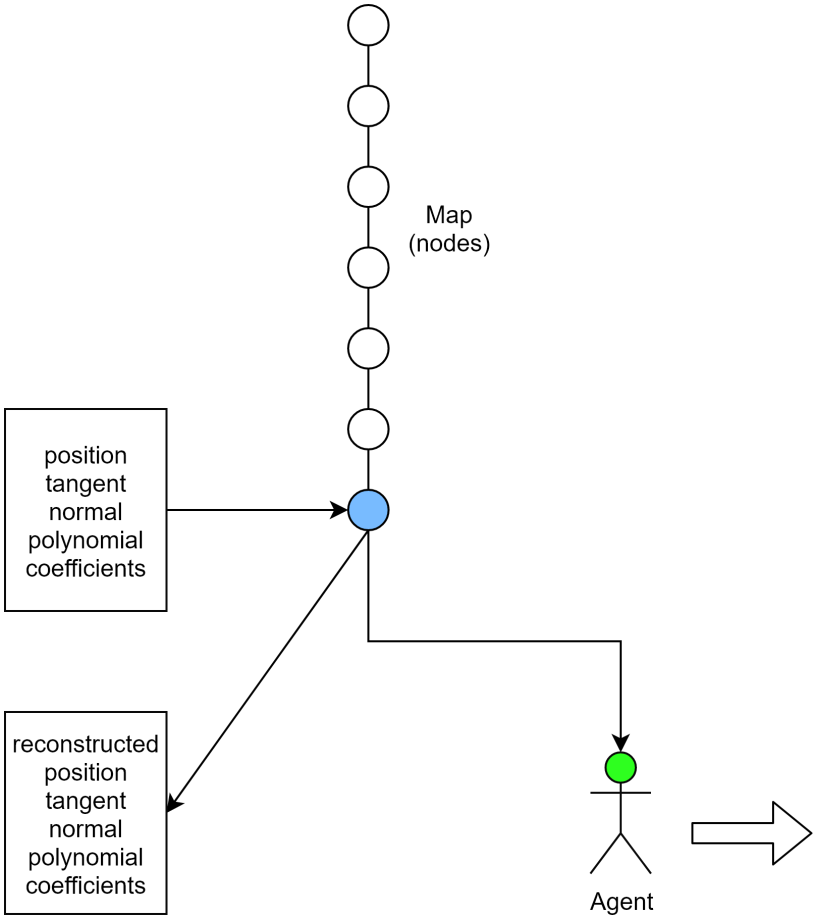


Full GAN Structure



b

Map Encoding



Input Encoders

Two types of encoders -- scene-global and agent-specific.

- Coordinate encoder -- nonlinear fully connected NN. Each node represents an agent, edges (i,j) represent updates from agent j's previous position and state into agent i's state. Each node update is fed:
    - The results of edge processors concatenated with
        - (position, map, image) encoders, and
        - the agent type one-hot vector

  to update the next time step's node state. The nodes are represented by LSTMs, either a single LSTM for all agent types, or an LSTM pr agent type (turned off by default).
- Map encoder
    - Map node encoder -- inputs: positions, element type (e.g., lane center line, crosswalk boundary, road boundary), tangent/normal, polynomial representation (total of 7 inputs). Output: same inputs, intermediate output for predictor.
    - Vectorize -- group input map coordinates into vectors by their ids. Obtain vector features via max-pooling.
    - Graph -- perform cross attention between vector features and agent features to obtain map-attended agent features.
- Position encoder -- coordinates are fed via a fully connected network to the LSTM.

Training Loss / Costs Structure

The following costs are used in training the predictor.

Generator Costs

The cost used for training the generator is a sum of the following terms (currently incomplete):

- General generator costs (`compute_generator_cost` in predition_model.py)
    - MoN_error (weighted via --mon-term-coeff): Minimum square error (minimized over all the predicted samples).
    - robust_error (weighted via --l2-term-coeff): Basically a truncated l2 error. Not used by default.
    - discriminator term (weighted via `--discriminator-term-coeff`) obtained directly from `discriminate_trajectory`.
    - weighted_loss_posterior (weighted via --weighted-l2-term-coeff): A weighted L2 error where the weights are based on an exponentiated magnitude of that very error. Not used by default.
    - discrete_loss (Computed in compute_hybrid_losses_with_unknown_labels, which is only executed if `--use-hybrid-outputs` is set. For pedestrian prediction this is not set by default.): Trying to make the sample weights match the exponent of the negative prediction errors via KL-Divergence.
    - acceleration_cost (`TrajectoryRegularizationCost` in `training_utils.py` if `--trajectory-regularization-cost` is set, which is by default not the case).
- Additional costs from the model (`generate_trajectory` in prediction_model.py)
    - Scene-specific additional costs (`fuse_scene_tensor` in prediction_model.py), which is currently not set.
    - Agent-input additional costs (`fuse_agent_tensor` in prediction_model.py) all coming from `agent_input_encoders`. In our default training setting, the losses from these input encoders are:
        - Map costs from (`MapPointGNNEncoder` in prediction_util_classes.py where we use the simple map encoder structure by default). These contain:
            - position_costs (weighted via --`map-reconstruction-position-coeff`)
            - tangent_costs
            - type_costs
        - Scene and agent image costs (`ImageEncoder` in prediction_util_classes.py), which are currently not set.

Discriminator Costs

We use the negation of the generator-discriminator loss to train the discriminator, but feeding both generated and real trajectory samples for training, as standard in predictor GANs, e.g., SocialGAN.

Module Documentation

Training Script

Each predictor (e.g., the one used for language-based or hybrid) should define its own training script. For example, there is a `train_pedestrian_trajectory_prediction.py` script for the generic training

settings and a `train_language_trajectory_prediction.py` for the language specific predictors. The script is organized as follows:

- Add required flags for the predictor: `parse_arguments` method has an additional_arguments_setter argument where you can add the additional setters to the list.
- Add predictor specific callbacks to the param set. For example, add "`additional_structure_callbacks`" to compute additional costs, and "`additional_trainer_callbacks`" to include the callbacks functions that are used at different places of the trainer.
- Finally, perform training by calling `perform_training_schedule` with the defined parameters and the created transforms.

Refer to `intent/multiagents/README.md` for examples of running commands with the scripts.

Pre-training / Curriculum Learning

Pre-training is done by defining a set of parameter sets, each for a slightly more complete model structure / modified costs, and partially training the model. Note that the models have to be compatible (e.g., stub encoder vs really encoding images -- the LSTM input width has to be the same and it's up to the user/developer to verify this). For more details, see the function `define_pretraining_parameter_sets` and the flags --pretraining-mode, --pretraining-definition, --pretraining-timescale, --pretraining-relative-lengths.

Evaluation Runner

The runner is invoked via intent/multiagent/run_pedestrian_trajectory_prediction.py. It collects the statistics over the test dataset beyond during-training statistics, include prediction example bird-eye views (BEVs), Refer to `intent/multiagents/README.md` for examples of running commands with the scripts, and Metrics Computation for more details.

Metrics Computation

Metrics currently computed in prediction_metrics.py:

- Finite Displacement Error (FDE) -- Take ground truth vs predicted trajectories at each time point, average of MoN over all samples.
- Average Displacement Error (ADE) -- Average the FDE over all time points.
- MoN (Minimum of N) -- minimum error over N samples (possibly squared error to ensure positive value)
- Miss Rate -- If the FDE at each time horizon is greater than the threshold corresponding to that horizon, it is considered a miss.

FDE, ADE, and Miss Rate are logged while training and displayed on Tensorboard or Weights and Biases. All metrics are computed during evaluation in the SaveErrorStatistics logging handler. After an evaluation run finishes, all the plots of values are saved as png images (default location is ~/intent/data_runner_results/).

- If the ground truth trajectory doesn't have enough horizon, "partial" results will use the latest available timestamp in that horizon and non-"partial" results will have "NaN" values to represent horizons outside the trajectory timestamps.

Convert to Prediction Protobuf

All of our data, either Argoverse, Waymo, or internal, were converted to prediction Protobufs. The definition of the protobuf is tricepts/tricepts/protobuf/prediction_training.proto
The conversion scripts are in data_sources/.

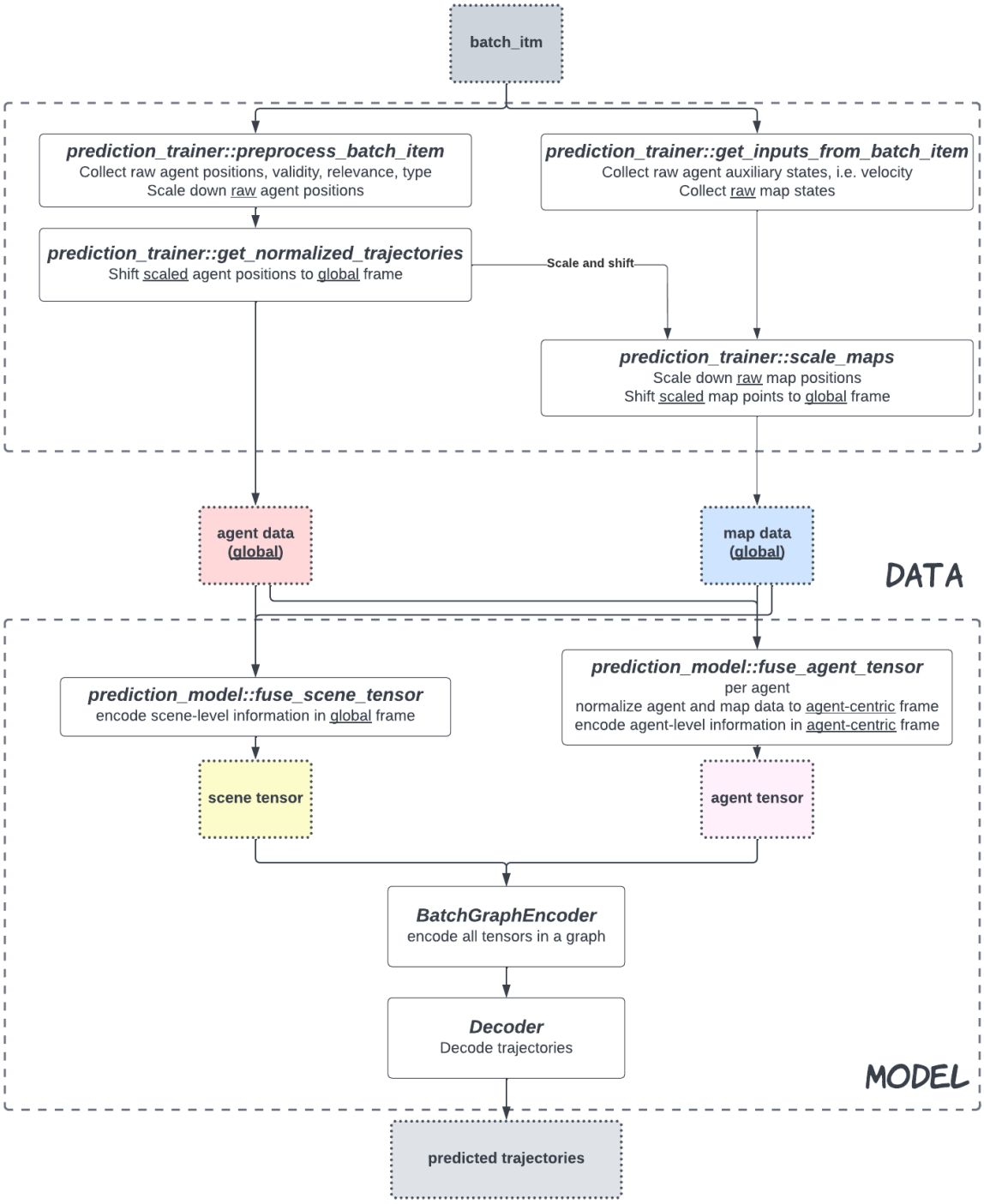
We assume one scenario per prediction protobuf.
There's two big chunks of data in a prediction protobuf.

- Trajectory and map
- Trajectory
    - Agents:
        - Ego agent
        - relevant agents:
            - the agents to predict
            - Argoverse: 1 max Waymo: 8 max
        - object of interest (two interactive agents)
            - Waymo exclusive
            - protbuf tag: ADDITIONAL_INPUT_INTERACTIVE
            - Mapped to relevant_agents in batch_item
        - Additional agents
            - Context agents, not used for training
            - protobuf tag: ADDITIONAL_INPUT_ADDITIONAL_AGENT
    - Agent types
        - Argoverse: only has car
        - Waymo: car, pedestrian, bicycle
- Map
    - Argoverse: center line only
    - Waymo: Lane line, boundary, zone, neighbors/connectivity, traffic lights
        - Traffic lights, neighbor/connectivity are not used

Pytorch Data Loader / Dataset Class

Data is loaded from the protobuf via a Pytorch dataset / data loader. The loader includes data transformations ("handlers") such as:

- Trajectory pre-processing -- via a callback. This allows smoothing and interpolation of the trajectories, as well as checking and invalidating bad examples.
- Map transformation -- via a list of callbacks. We support both raster-based and point-based / graph-based map encodings.
    - Point-based handler -- Points and their local features (tangent/normal, optionally polynomial coefficients) are sent based on the polyline representation.
    - Global frame: Downscale then shift to relevant agents trajectories' center of mass



    - Raster-based handler -- Protobuf polylines are rasterized into an image with specified resolution and range.
- Image transformation handlers -- allow cropping, resizing and caching of the transformed images.

Decoder

The decoder we use is an LSTM-based decoder. We use the LSTM's hidden state to emit coordinates and feed the output back into the next state as in SocialGAN. We can also generate latent factors (see below) and feed their state at given timepoints into the decoder, allowing us to do a set of different temporal/social structures, such as attention-based predictors and hybrid predictors.

Latent Factors

Latent factors are structured representations that help the decoder learn the meaningful state transitions and agent associations. They can be in the form of temporal and inter-agent attention, explicit duration modes, hybrid automata, or language tokens. The interface for latent factors is defined as the `LatentFactors` class. Each type of latent factors needs to implement several functions:

- `generate_initial_factors`: This is called before making predictions for each time step. It takes the initial states output from the encoder to generate the initial factors if needed.
- `time_sample`: At each time step, the decoder calls this function with the current agent state to update agents' states with the sampled state contribution tensor (e.g., the tensor computed from attentions).
- `predict_agent_state`: Can be used to update the predicted agent state, or store per-timestep information (e.g., for temporal attention).

We do not instantiate a particular type of latent factors in the decoder. All latent factors classes should be added in the training script, populated as "`latent_factors`" in the `param` dictionary (see `train_language_trajectory_prediction.py`,`train_hybrid_vehicle_prediction.py` for examples), and are activated, e.g. in the decoder class. When the decoder sees "`latent_factors`" is populated, it will execute the above two functions in `decode`.

Trainer

The trainer is implemented in the `PredictionProtobufTrainer` class in `prediction_trainer.py`. The main method of this class is `train()`. It is responsible for creating the DataLoaders for the respective datasets, running the entire training process, and interfacing with the logging classes via the logging interface. This class is designed for a generator / discriminator structure (although the discriminator step can be omitted). The optimization steps for the generator and discriminator are executed in `optimize_generator()` and `optimize_discriminator()` respectively.

Caching

- The full dataset could be cached into a set of pickle and image files, which drastically reduce the size of the space used to store the data, speed up the data accessing/processing and thus speed up training.
- Cache versioning
    - The cache is versioned and hashed so the cache is unique to each input file.
        - The version number is defined as `DATA_AND_HANDLER_VERSION`.
        - Note: It must be **manually incremented** every time data related code is changed.
        - Failure to do so will result in crashes.
    - Input data and thus cached data are affected by some parameters of the training
        - For example, `--past-timesteps` the number of past trajectory points used to do prediction or `--scene-image-timepoints` the frame number of the agent images to use.
        - Those parameters are captured and hashed to form part of the cache version.
        - There's hash for main parameters and for each individual data handlers
        - A list of parameters used to determine main cache hash is in `PARAM_HASH_LIST` in `cache_utils.py`.
    - Example of a cache's name
        - data_ver_4-individual-param_3df915515abc05dd6b02cecec701b6d1
        - Version modularity parameter's hash
    -
- Generate cache

    - **Eager**, use intent/multiagents/run_validation.py [--param (use the exact same parameters you would use to run training, since some parameters would change who the data is generated)] --cache-dir ~/your_cache_dir
    - **Lazy** instantiation. Just run training, it will cache dataset to a default cache dir
- What is cached:
    - trajectories
    - Cropped/resized images (global and agent)
    - Map (points or image)
- Cache auto downloading
    - sync_model_to_s3 to auto download the cache from s3
    - It is enabled when --cache-dir is pointing to a dir that doesn't exist **and** --cache-read-only true
    - --dataset-names will determine which s3 files to download

Logging Handlers

Logging handlers are implemented as callback classes that log helpful information during validation during each iteration of an epoch and at the end of each epoch. SaveErrorStatistics and WorstCaseLoggerHandler are the handlers that create the data referenced the most when evaluating the model performance.

SaveErrorStatistics

SaveErrorStatistics logs error metrics calculated for each sample of each instance in the testing dataset. Finite Displacement Error (FDE) and Pedestrian Traversal are the primary error metrics calculated in this logger.

WorstCaseLoggerHandler

This logging handler saves a Bird's Eye View(BEV) image and the values for our metrics and evaluation criteria from the worst performing instances for each of our costs.

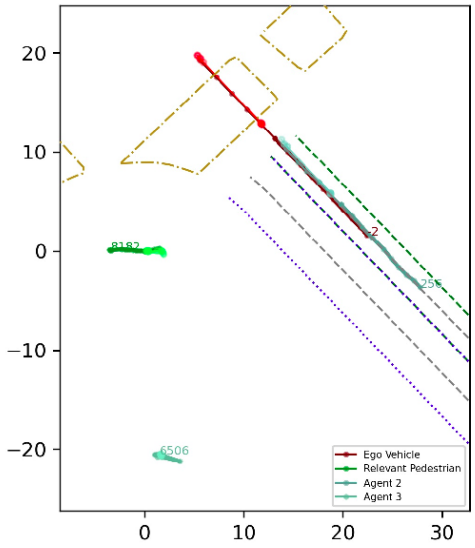
Visualization

There are several visualization locations in the training stack:

- Data reharvesting phase
- Training pipeline
- Evaluation runner

The data reharvesting phase uses its own visualization functions, aimed more at validating data correctness and usual artifacts mitigation in raw data (e.g., tracker noise).

The training pipeline can log results in either Tensorboard or Weights & Biases via two child classes, `TensorboardTrainingLogger` and `WandbTrainingLogger`, respectively. For visualization prediction examples we use the function `visualize_prediction`.



Trajectory:

Dark color is the past trajectory, lighter color is the predictions.

Map:

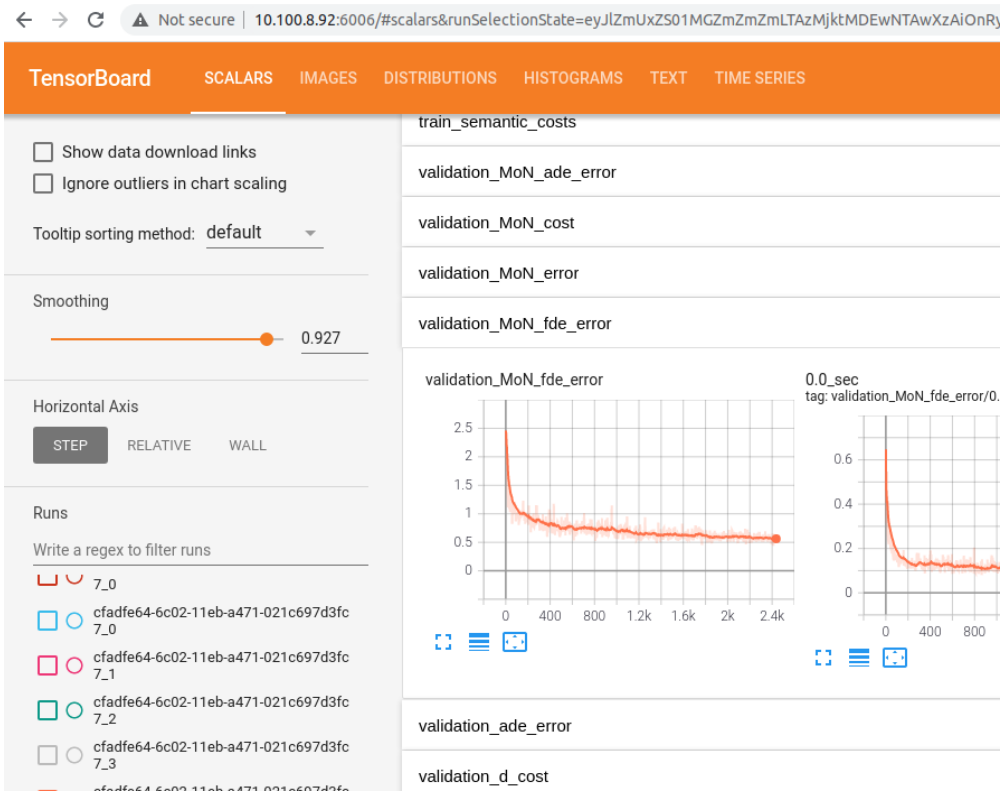
Center lane line: Gray
Right lane line: dark green
Left lane line: blue
Crosswalk: Dark gold
Lane boundary line: dark blue

Available Tensorboard Statistics

Start tensorboard on EC2 node on a separate screen with

```
tensorboard --logdir ~/intent/logs --bind_all
```

To access: use `http://<computer ip address>:6006/`



We can also use Weights and Biases logging wrappers.

Main tabs:

- Scalars - scalar plots over the convergence of the model
- Images - example images
- Text - collected text items such as source tlogs for examples, model and training parameters, etc

Scalar Categories

`Memory_usage`	Memory usage in GB
`train_MoN_ade_error`	MoN ADE error for relevant agents
`train_MoN_fde_error`	MoN FDE error for relevant agents
`Train_data_cost`	The data cost for the training
`Train_g_residual`	Overall residuals of the parameter updates of the generator -- norm of change over time.
`validation_MoN_ade_error`	MoN ADE error for relevant agents for validation set
`validation_MoN_fde_error`	MoN FDE error for relevant agents for validation set
`Validation_d_cost`	Discriminator cost for the validation set

Image Entries/Tab

- Legend for all birds-eye views of prediction results:
    - Thick dark solid line -- past trajectory
    - Thin solid lighter line -- future acausal / ground-truth trajectory
    - Thin transparent lighter lines -- predictions

    - Red, green -- ego-vehicle and pedestrian, respectively
- Main tabs:
    - vis -- visualization (validation set), train -training set
    - validation_worstcase_X -- validation errors
    - map_position_X -- shows the reconstruction of the map from the map encoder (blue vs red). If the points do not align, it means that the map encoder is still converging, probably for prediction as well as reconstruction.

Text Entries/Tab

- commit -- the commit used in the risk_aware_driving repo
- training_schedule_key -- the phase it was in in the training schedule (e.g. gnn_edges)
- sys.argv -- the arguments used for running the training script, as sys.argv
- sys.argv_string -- the command for running the training as a string (not a list)
- params -- the model parameters / command line arguments and flags
- parameters of the model -- the model's weights sizes, all of the tensor dimensionalities of weights
- dataset length
- semantic label weights