# Multimodal Trajectory Prediction in Multi-Agent Scenarios

Jan Duscherer  Lukas Röß

Seminar: Video Analysis and Object Detection

Lecturer: Prof. Dr. Claudius Schnörr

June 28, 2025

# Contents

# List of Figures

# List of Tables

# 1 Introduction

## 1.1 The Autonomous Driving Stack

Safe and efficient navigation in autonomous driving hinges on a multi-stage pipeline of *perception*, *prediction*, *ego-planning* and *control* [1]. The perception module processes raw sensor streams such as LiDAR, radar, and multi-view camera data to produce a rich representation of the scene, including the (kinematic) states of all traffic participants (*agents*, i.e., vehicles, pedestrians, cyclists) and static map elements like lane markings, traffic lights and pedestrian crossings [2]. This vectorized scene-representation serves as the input to the *motion forecasting* module, which is tasked with inferring the future trajectories of all agents in the scene over some planning horizon. The output is not a single, deterministic path, but a probabilistic, multimodal distribution over possible futures of each agent, which enables the pro-active planning of feasible and safe maneuvers and ego-trajectories from which the control module can derive the necessary vehicle commands.

## 1.2 Challenges in Multi-Agent Motion Forecasting

Despite impressive advances, three interrelated challenges limit current motion forecasting models The behavior of traffic participants is inherently *non-deterministic* and involves highly complex interactions with the environment and other agents. This necessitates models that can capture these complex dynamics and produce diverse, probabilistic motion forecasts to account for the wide range of possible future behaviors, whose uncertainty grows exponentially with the planning horizon as errors compound over time. Attention-based architectures have emerged as powerful tools to model these complex interactions and overcome the issue of *mode collapse* that plagued earlier approaches. However, traditional *agent-centric* encoding schemes require expensive per-frame re-normalization, preventing the caching and reuse of previously computed features and hindering real-time performance[3]. Third, *scalability and generalization* remain elusive: models trained on a single dataset like nuScenes, Argoverse 2, and Waymo [4, 5, 6] often fail to transfer across benchmarks with disparate formats, sampling rates, and map semantics [7].

## 1.3 Contributions and Report Structure

We begin in section 2 by reviewing key forecasting paradigms and explore the query-centric encoding scheme [8] as a *geometrically* elegant solution towards *joint motion forecasting* and *streaming inference* in autonomous driving in subsection 2.2.
We then introduce the `UniTraj` framework [7] in section 3, which provides a

unified interface for single-agent trajectory prediction across multiple datasets. Here, we describe the practical contributions of this seminar work.
In section 4, we introduce `UniTraj`'s data harmonization pipeline, which converts heterogeneous datasets into a single, agent-centric format.
Building on the insights established in section 2, we then provide in-depth reviews of four selected motion-forecasting models in section 5: subsection 5.1 introduces a conceptually simple raster-based approach, the `CASPNet` [9]. Following this, subsection 5.2 presents the `CASPFormer`, a hybrid CNN-transformer architecture that extends the raster-based approach to incorporate attention mechanisms for improved feature extraction and interaction modeling. Next, subsection 5.3 discusses the `MTR` [10], another agent-centric model which was implemented within the `UniTraj` framework. Finally, subsection 5.4 presents the LMFormer [2], a query-centric model that maintains various invariances throughout the entire pipeline.
In section 6 we describe the experimental design and results of training the `MTR` within the `UniTraj` framework and validate the results with those reported in the `UniTraj` paper [7].

# 2 Theoretical Background & Related Work

**Notation.** We denote $T_p$ and $T_f$ as the numbers of observed and predicted timesteps, respectively. Following UniTraj conventions [7], agent trajectories are represented as $\boldsymbol{X}_d \in \mathbb{R}^{N_{\max} \times T_p \times F_{ap}}$ and map polylines as $\boldsymbol{X}_s \in \mathbb{R}^{K_{\max} \times L \times F_{map}}$, where $N_{\max}$ is the maximum number of agents, $K_{\max}$ is the maximum number of map polylines, $L$ is the points per polyline, and $F_{ap}, F_{map}$ are the respective feature dimensions. Ground truth trajectories for the center agent are denoted $\boldsymbol{y}_c \in \mathbb{R}^{T_f \times 4}$. For rasterized approaches, BEV representations use $H \times W$ spatial resolution with $F_d, F_s$ channel dimensions for dynamic and static inputs, respectively. Transformer models employ $M$ output modes, $K_s$ sampling points per deformable attention query, and $N_h$ attention heads. Feature pyramid networks utilize $L_{\text{FPN}}$ levels indexed by $\ell \in \{0, \ldots, L_{\text{FPN}} - 1\}$, with feature maps $C_\ell \times H_\ell \times W_\ell$ at each level. A comprehensive symbol table is provided in Appendix A.1.

## 2.1 Scene Representation Paradigms

Scene representations translate outputs of the perception stage into a tensor that subsequent neural modules can exploit. Desirable properties include:

(i) high geometric fidelity

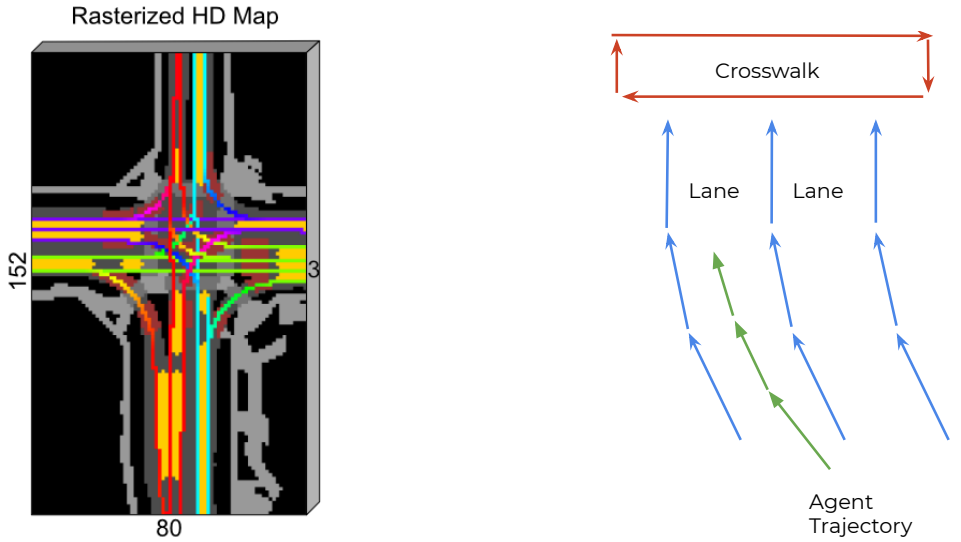
(ii) invariance to global transformations (translation, rotation, time-shift) $SE(2) \rtimes \mathbb{R}$

(iii) information density, ensuring that representations encode all relevant properties of the scene without unnecessary redundancy

(iv) suitability for efficiently modeling spatio-temporal, kinematic, semantic, and topological relationships between scene elements

(v) computational re-use across frames [8, 2].

The choice of scene representation fundamentally affects how effectively the predictor can capture essential relationships and dynamics in complex traffic scenarios, and hence it's capacity to produce accurate and diverse motion forecasts. The *rasterized* and *vectorized* paradigms represent the two main approaches to scene representation in trajectory prediction, as illustrated in Figure 1.




(a) Rasterized BEV encoding

(b) Vectorized polyline preservation

Figure 1: Scene representation paradigms in trajectory prediction: (a) rasterized approaches stack agent trajectories and HD maps into BEV images [9], (b) vectorized methods preserve geometric polylines [11].

**Raster grids.** Early systems stack past agent masks and HD-map layers into $F$-dimensional BEV (Bird's Eye View) images, leveraging convolutional backbones to capture spatial relationships while keeping runtime independent of the number of agents [12, 13]. Specifically, these approaches construct: (i) a BEV stack of past agent trajectories $\mathbf{I}_d \in \mathbb{R}^{T_p \times H \times W \times F_d}$ and (ii) a static HD-map raster $\mathbf{I}_s \in \mathbb{R}^{H \times W \times F_s}$.
The CASPNet family of motion forecasting models, consisting of the original CASPNet [9], which utilizes a fully convolutional architecture, CASPNet++ [14], and the CASPFormer [15], exemplifies interesting architectural choices within this paradigm and will be discussed in greater detail in subsection 5.1.
However, rasterized approaches suffer from limited geometric fidelity and redundant pixel information due to grid-based discretization of scene elements' geometric and kinematic properties [2]. Additionally, respecting agent identities is infeasible as it would require separate channels per agent, introducing further redundancy; CASPNet++ [14] addressed this using single BEV per agent. Furthermore, rasterized approaches allow only shared coordinate systems, which is suboptimal for leveraging geometric isomorphisms [8].

**Vector representations.** Later work encodes agents and lanes as vectorized geometric primitives such as polylines, enabling graph (LaneGCN [16], VectorNet [17]) or transformer based (QCNet [8], QCNeXt [3], LMFormer [2])

approaches with higher geometric fidelity but runtime that grows with scene complexity. These representations are more compact, preserve higher geometric fidelity, and enable explicit modeling of complex spatio-temporal and social relationships between scene elements. Lane information uses two main representations:

- **Point-based**: Each polyline $L_p^i = [P_1^i, P_2^i, \ldots, P_K^i]$ with $K$ control points $P_k^i$ [17, 18].
- **Segment-based**: Converts to $L_v^i = [V_1^i, V_2^i, \ldots, V_{K-1}^i]$ where $V_k^i = [P_k^i, P_{k+1}^i]$ stores lane segment vectors. This explicitly encodes road curvature [16, 18, 8].

Agent trajectories use analogous representations:

- **Trajectory points**: $\mathcal{T}_{in}^a = [P_1^a, P_2^a, \ldots, P_T^a]$ with global positions $P_t^a$.
- **Motion vectors**: $M_t^a = [P_2^a - P_1^a, \ldots, P_T^a - P_{T-1}^a]$ derived from trajectories, representing the displacement between timesteps [2].

Vectorized approaches employ either *agent-centric* coordinate systems (all scene elements normalized to a single ego-centric frame) or *query-centric* paradigms. The choice fundamentally affects computational efficiency, invariance properties, and multi-agent reasoning capabilities, with query-centric approaches offering significant advantages for streaming applications and parallel multi-agent prediction [8].

UniTraj [7] employs a vectorized and agent-centric representation, representing both agent trajectories as vertex lists and map polylines as uniformly sampled list of vertices.

## 2.2 The Query-Centric Paradigm

**The short-comings of agent-centric approaches.** While the agent-centric paradigm is a conceptually simple solution that aligns well with the historical focus on single-agent prediction, where the entire scene is expressed in a global frame centered on the ego vehicle, it is not well-suited for the emerging task of *multi-agent* motion forecasting as it yields roto-translation and time-invariance for only the center agent [2].
Furthermore, this approach becomes computationally infeasible when utilized within the framework of *factorized attention*. While typical encoding strategies, such as those employed in the CASPNet family, squeeze the entire temporal dimension, and subsequently apply agent-map and agent-agent fusion on this time-invariant representation, factorized attention maintains separate spatio-temporal latent representation of all entities in the scene, allowing the model to capture more complex spatio-temporal relationships, such as the interactions between mutliple agents over the course of multiple

timesteps. However, this implies cubic complexity for each fusion step:

$$\begin{aligned} \text{Temporal:} \quad & \mathcal{O}\big(N_{\max} T^2\big) \\ \text{Agent}\leftrightarrow\text{Map Fusion @ } t\text{:} \quad & \mathcal{O}\big(N_{\max} T K\big) \\ \text{Agent}\leftrightarrow\text{Agent Fusion @ } t\text{:} \quad & \mathcal{O}\big(N_{\max}^2 T\big) \end{aligned} \tag{1}$$

where $N_{\max}$ is the maximum number of agents, $T = T_p + T_f$ the number of total timesteps, and $K$ is the number of map polylines. This cubic complexity arises from the need to compute pairwise interactions between all agents and map elements at each timestep, leading to significant computational overhead in dense traffic scenarios.

**Query-centric encodings** The query-centric paradigm aims to maintain the representational capacity of factorized attention, while reducing the inference latency. The most significant downside of the agent-centric paradigm in this context is that it requires the entire scene to be re-encoded at every timestep, as the location and heading of the target agent change. This means that the model must recompute the entire scene representation for each target agent at *each timestep*, leading to a significant incr6ase in computational complexity and latency, especially in scenarios with many agents and long observation windows [8].

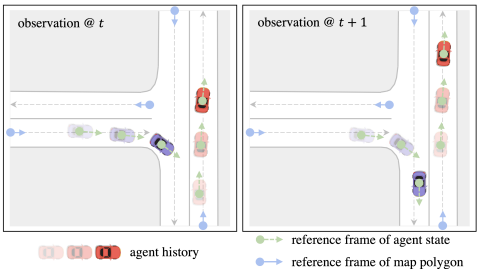



Figure 2: [8] Query-centric encoding: each scene spatio-temporal entity lives in its own local coordinate system

Instead of expressing the scene in a single, temporally-evolving global frame, the query-centric paradigm establishes a *local coordinate system* (or *fiber*) for each scene element, in which all spatio-temporal and kinematic properties are expressed. This becomes especially handy, when considering that motion forecasting is inherently a *streaming* task during inference: whenever a new observation arrives, the oldest observation is dropped and the newest one is added to the queue. Hence, two temporally adjacent scene representations share $T-1$ overlapping timesteps, which can be cached and reused in the

next step. This results in a significant reduction in computational complexity, lowering the overal runtime for each kind of fusion step by $T$ from the cubic complexity of the agent-centric paradigm to quadratic complexity:

$$\underbrace{\mathcal{O}(N_{\max}T^2) + \mathcal{O}(N_{\max}TL) + \mathcal{O}(N_{\max}^2 T)}_{\text{standard factorized attention in an AC-paradigm}} \longrightarrow \underbrace{\mathcal{O}(N_{\max}T) + \mathcal{O}(N_{\max}L) + \mathcal{O}(N_{\max}^2)}_{\text{query-centric streaming}}$$

### Local frame construction

The query-centric paradigm builds a local spacetime frame for each scene element, creating what is commonly referred to as *fibres*, which can be seen as slices through the global scene manifold. The concept of fibres and more generally *fiber bundles* is borrowed from differential geometry, where a fiber bundle is a structure that consists of a base space and fibers above each point in the base space, allowing for local coordinate systems to be defined independently at each point[19].

With this distinction in mind, we now describe how specific local coordinate frames are constructed for different scene elements:

**Agent states.**
For the $i$-th agent at timestep $t$, the local frame is anchored at the agent's spatial position $\mathbf{p}_i^t = (p_{i,x}^t, p_{i,y}^t)$ with the $x$-axis aligned to the agent's instantaneous heading $\theta_i^t$ [2]. The transformation from global to local coordinates is:

$$\mathbf{x}_{\text{local}}^{(i,t)} = \mathbf{T}_{i,t}^{-1}\, \mathbf{x}_{\text{global}^{(i,t)}}, \qquad \mathbf{T}_{i,t} = \begin{bmatrix} \cos\theta_i^t & -\sin\theta_i^t & p_{i,x}^t \\ \sin\theta_i^t & \cos\theta_i^t & p_{i,y}^t \\ 0 & 0 & 1 \end{bmatrix} \in \mathrm{SE}(2). \quad (2)$$

This yields $N_{\max} \times T$ distinct fibres over the observation window, where each agent state $(\mathbf{p}_i^t, \theta_i^t, \mathbf{v}_i^t)$ defines its own reference coordinate system and $\mathbf{x}_\bullet^{(i,t)} \subset \mathbf{X}_d$ represents some arbitrary spatial or kinematic vector (i.e., motion vector, velocity, ...) belonging to the $i$-th agent at timestep $t$.

**Map elements.**
Lanes are represented as segment lists. Each segment's start vertex acts as the origin and the direction of the first segment defines the $x$-axis, leaving only the (normalized) length as an explicit feature [2].

Each element's geometric attributes (velocity vectors, motion trajectories, or sampled map points) are then expressed in polar coordinates relative to their respective local frames and lifted into a higher dimensional space using encoding schemes like (learnable) Fourier features [8, 20]. These encodings can then be concatenated with the semantic attributes (e.g., agent type).

**Relative descriptors and embeddings.** Having established the conceptual framework of fibres and local coordinate frames, we now describe

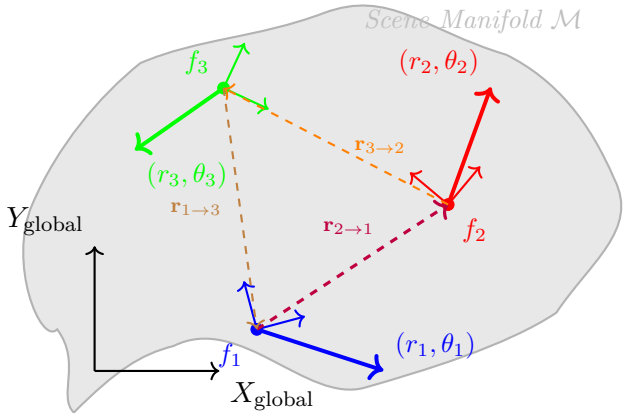



Figure 3: Query-centric scene representation as fiber bundle: Each scene element defines a local fibre $F_i$ with its own coordinate system $(x_i, y_i)$ embedded in the global scene manifold $\mathcal{M}$. The colored orthogonal arrows represent specific local coordinate frames $f_i \in F_i$ chosen from each fibre. Motion vectors $(r_i, \theta_i)$ are expressed in local polar coordinates, while relative descriptors $\mathbf{r}_{j\to i}$ encode the 4D spatial-temporal relationships between fibres, preserving distance, bearing, orientation difference, and temporal offset.

how spatial-temporal relationships between scene elements are encoded. The query-centric paradigm constructs relative positional embeddings that preserve all essential geometric information while maintaining invariance properties. For any pair of scene elements with absolute tuples $(\mathbf{p}_i^t, \theta_i^t, t)$ and $(\mathbf{p}_j^s, \theta_j^s, s)$, QCNet [8] forms a 4-dimensional descriptor:

$$\mathbf{r}_{j\to i}^{s\to t} = \left[\|\mathbf{p}_j^s - \mathbf{p}_i^t\|_2,\ \text{atan2}(p_{j,y}^s - p_{i,y}^t,\ p_{j,x}^s - p_{i,x}^t) - \theta_i^t,\ \theta_j^s - \theta_i^t,\ s - t\right]. \quad (3)$$

This descriptor fully preserves the spatial-temporal offset between elements and is designed to be invariant under global SE(2) transformations. Each component encodes a specific geometric relationship:

(i) The Euclidean distance $\|p_j^s - p_i^t\|_2$ captures the spatial separation between elements.

(ii) The angular offset $\text{atan2}(\cdot) - \theta_i^t$ represents the bearing from element $i$ to element $j$ in $i$'s local coordinate frame.

(iii) The orientation difference $\theta_j^s - \theta_i^t$ encodes the relative heading between elements.

(iv) The temporal offset $s - t$ preserves the time relationship between observations.

Following QCNet [8], this 4D descriptor is lifted into a high-frequency

representation via Fourier features. These embeddings are then concatenated with other features to form the input sequences that will be processed by the decoder.

### A geometric perspective.

The query-centric paradigm exploits three fundamental symmetries that reflect the underlying physics of trajectory prediction. By placing each scene element (agent or map polygon) in its own local $SE(2)$ coordinate frame, this approach dramatically simplifies the learning problem compared to agent-centric methods by theoretically allowing a decomposition of the global scene manifold into a composition of local fibres, each representing a standardized set of spatial and kinematic properties or relatively simple connections between different fibres.

**Permutation invariance.**
Agents and map elements form an unordered set—no element should be privileged. Query-centric approaches preserve this natural symmetry by treating all elements identically through local frames, while agent-centric methods break symmetry by choosing one agent as the global origin. Formally, for any permutation $\sigma$ of scene elements, the encoding remains equivariant: $f(\sigma \cdot S) = \rho(\sigma) f(S)$.

**SE(2) invariance.**
Rigid translations or rotations of the entire scene must leave the encoding unchanged:

$$f_{\mathbf{\Psi}}\big(g \cdot S\big) = f_{\mathbf{\Psi}}(S) \qquad \forall\, g \in SE(2). \tag{4}$$

Because every geometric attribute is expressed either in its *own* local frame or as a *relative* descriptor between two frames, dependencies on the global origin and its orientation never appear. $f_{\mathbf{\Psi}}$ might represent solely the encoder network, or in case of the LMFormer [2] the entire architecture including the decoder.

**Temporal translation invariance.**
Motion forecasting uses sliding time windows—shifting the time origin should not affect predictions. For any time shift $\tau \in \mathbb{R}$, the representation satisfies $f(S(t+\tau)) = f(S(t))$. Query-centric methods enable efficient caching since relative temporal descriptors remain unchanged as windows slide. Agent-centric approaches violate this property because their global reference frame evolves with the target agent, requiring full re-encoding at each timestep.

Viewed through the lens of *fiber bundles*, the global scene manifold factorises into many almost-identical *fibres*—one for every agent state or lane segment—each parameterised by a small, standardised set of spatial and kinematic variables. Because all agents obey the same motion constraints and all map segments share the same geometric primitives, these fibres are

essentially isomorphic.
Consequently, we hypothesize that the model can split its representational capacity to learning the *relations between fibres*, as well as the relatively *simple structures of the fibres* themselves instead of having to learn a non-decomposable global scene manifold. The pairwise encodings of the relative descriptors *connecting* the fibres inhabit an even simpler manifold, further shrinking the hypothesis space. This decomposition supplies the key inductive bias of the query-centric paradigm: by respecting permutation symmetry and $\mathrm{SE}(2) \rtimes \mathbb{R}$ invariance, it breaks the otherwise intractable global scene manifold into a set of small, repeatable sub-manifolds connected by simple relations, yielding representations that are both more data-efficient and more interpretable than those produced by agent-centric encodings.

**The downside of query-centric encodings.** Query-centric (QC) models trade *compute* for *memory*. Because every agent state and map segment owns a persistent token for *each* timestep and because all pairwise keys/values are cached to enable streaming reuse, the scene tensor grows linearly with $N_{\max} T$ and can reach hundreds of megabytes for a single batch. QCNet's public implementation reports[1] that training a *single* model on Argoverse2 requires $\sim$160GB of GPU VRAM, typically split across 8 RTX 3090 24 GB cards. Even on inference, caching the key-value memory for a 10 s sliding window with 128 agents can exceed 10 GB. However, this can be dealt with by not caching the fully embedded key-value tensors, but rather the raw relative descriptors and local frames and computing the embeddings on-the-fly during training and inference as done in [2]. Furthermore, the QC paradigm can be combined with sparse attention mechanisms, such as *deformable attention* as explored in subsection 5.2 to further reduce the memory footprint.
We can safely conclude that the huge memory cost in [8] mostly stems from their complex architecture and the caching rather than the query centric paradigm.

# 3 The Need for Unification

This section presents an overview of the need for a unified framework for trajectory prediction in autonomous driving, addressing the challenges of dataset heterogeneity and the complexities of motion forecasting tasks. The discussion highlights the importance of standardizing data formats, feature characteristics, and preprocessing pipelines to enable effective model training and evaluation across diverse datasets.

[1] https://github.com/ZikangZhou/QCNet

## 3.1 Unification of Motion Forecasting Datasets

The `UniTraj` framework addresses the fundamental challenge of standardizing multiple motion-forecasting datasets that exhibit substantial heterogeneities in both *data formats* and *feature characteristics*. The former encompasses differences in data structure and organization, while the latter stems from variations in spatio-temporal resolution, coverage range, and semantic annotation schemes.

To overcome format discrepancies, `UniTraj` leverages `ScenarioNet` [21] for conversion into a common format, thereby eliminating the need for multiple preprocessing implementations. However, feature characteristics require additional harmonization. Temporal coverage varies substantially across datasets, with historical trajectories ranging from 1 second in `WOMD`[6] to 5 seconds in `Argoverse 2`[5], and future prediction horizons extending from 6 to 8 seconds. Map resolutions and semantic annotations differ significantly, with all datasets providing *scene-centric* HD maps at varying resolutions. Agent features also exhibit substantial variations: `nuScenes` provides only velocity and heading, while `WOMD` includes comprehensive 3D bounding box annotations. Notably, `Argoverse 2` provides bounding box and rich semantic annotations, but these are lost during `ScenarioNet` format conversion. The conversion and normalization process is outlined in subsection 4.1.

## 3.2 Unified ML Infrastructure

Beyond dataset unification, trajectory prediction research requires a comprehensive machine learning infrastructure that handles the complexities of modern deep learning workflows. The UniTraj framework addresses this need by providing essential components that allow researchers to focus on model innovation rather than infrastructure implementation.

These components encompass the complete research pipeline: training and evaluation frameworks, data handling and preprocessing modules, logging and visualization tools, shared metrics and loss functions, and experiment management systems that facilitate reproducibility and systematic experimentation. Such a framework should be designed with transparency and modularity as core principles, enabling seamless extension and integration of new datasets, models, evaluation metrics, and additional functionalities such as callbacks or custom training procedures.

## 3.3 Limitations of UniTraj

While the original UniTraj framework provided foundational functionality for unified trajectory prediction research, it suffered from poor software quality and architectural design, which limited its usability and transparency. As the goal of this seminar work was to understand the task of motion

forecasting from the ground up, we undertook a comprehensive refactoring of the UniTraj framework to address these limitations in order to achieve a holistic understanding of components belonging to the motion forecasting task.

Some of the most significant limitations of the original UniTraj framework, that motivated the refactoring, are summarized below:

- **Monolithic Architecture:** The complex data processing pipeline consisted of a single monolithic script with poor separation of concerns, making it almost impossible to understand data flow and transformations, extend individual components or fix issues that we observed in the resulting dataset.
- **Insufficient Type Safety:** Critical symbols lacked proper type hints, comments, and docstrings; the `Hydra` configuration system was used as an untyped dictionary without structural validation, such that it was difficult to work with these configuration objects, to understand the impact or expected values of parameters, and to catch potential issues early in the development process.
- **Data Split Integrity:** The framework failed to respect original train/-validation/test splits from source datasets, violating standard machine learning practices. Our efforts to resolve this issue turned out to be quit frustrating due to the intransparent path and file handling in the original implementation.
- **Inadequate Documentation:** Absence of comprehensive documentation and standardized coding practices made it challenging to work with and understand the framework.

**Infrastructure and Framework Integration Issues.** Despite utilizing modern frameworks like PyTorch Lightning, the implementation failed to follow recommended practices:

- **PyTorch Lightning Misuse:** Improper integration lacking `LightningDataModule` usage, inadequate logging functionality, no easy configuration of the components of this framework and improper integration with Weights & Biases.
- **Path Management Deficiencies:** Poor filesystem path handling without clear control over data sources, destinations, model checkpoints, and logs.
- **Missing Training Capabilities:** Absence of essential deep learning features including automatic mixed precision, gradient accumulation, learning rate scheduling, distributed computing support, and essential callbacks (learning rate monitors, early stopping, model checkpointing), all of which

are relatively easy to use if PyTorch Lightning is used properly.

### Revision of the UniTraj Framework

This section details the comprehensive refactoring applied to the original UniTraj framework to achieve a modular, type-safe, and extensible codebase that addresses all identified limitations while maintaining full functionality. The revised framework implements a hierarchical configuration-driven architecture, of which a small subset is visualized in Figure 4.

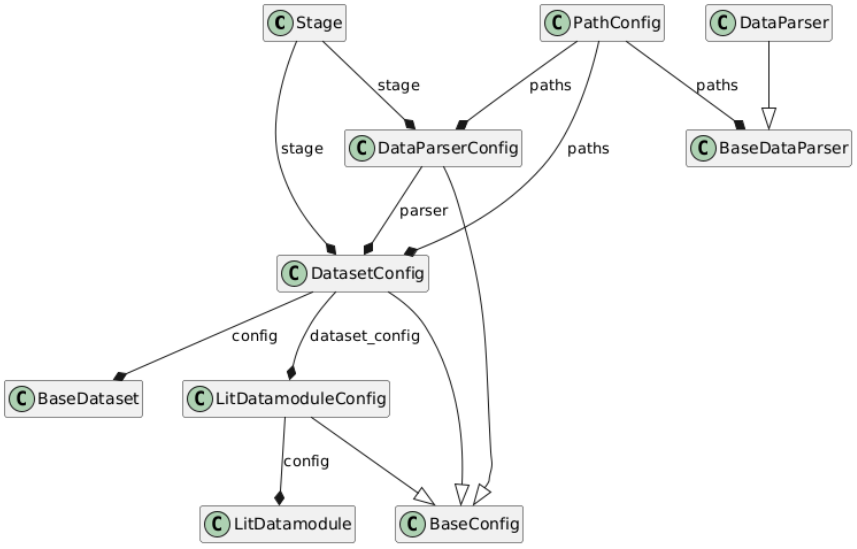



Figure 4: UniTraj data handling architecture showing the relationships between configuration classes, data parsers, datasets, and Lightning modules.

**Core Architectural Improvements.** The refactoring introduced three foundational design patterns that address the primary limitations of the original implementation:

- **Config-as-Factory Pattern:** Implemented through `BaseConfig` with strongly typed configuration classes for each functional component. Every configuration object serves as both a type-safe parameter container and a factory for instantiating its corresponding component.
- **Modular Data Processing Pipeline:** Decomposed the monolithic preprocessing script into three specialized stages with clear separation of concerns: `BaseDataParser` orchestrates high-level pipeline execution including multiprocessing, HDF5 storage, and metadata collection; `DataParser` implements a nine-phase preprocessing pipeline; and `BaseDataset` provides efficient PyTorch-compatible data loading.

- **Strongly Typed Interfaces:** Introduced comprehensive type definitions in `types.py` using `TypedDict` and Pydantic `BaseModel` classes, replacing untyped dictionary passing and enabling compile-time shape validation. Defines `RawScenarioDict`, `ProcessedDataDict`, `DatasetItem`, and `BatchInputDict` interfaces.
- **Lightning Data Module:** `LitDatamodule` provides standardized data loading with custom collate functions, configurable batch sizes, worker processes, and automatic train/validation/test split handling while maintaining original dataset split integrity.
- **Trainer Factory:** `LitTrainerFactory` enables easy `pl.Trainer` configuration with support for multi-GPU training, mixed precision, gradient accumulation, learning rate scheduling, and comprehensive callback management.

**Infrastructure and Monitoring Enhancements.** Comprehensive improvements to logging, experiment tracking, and development workflows:

- **Centralized Path Management:** `PathConfig` consolidates all filesystem paths with automatic directory creation and propagation to child configurations, eliminating scattered path management errors.
- **Hierarchical Configuration:** The `ExperimentConfig` serves as the top-level orchestrator with automatic parameter propagation and the `inspect()` method for complete configuration tree visualization, improving experiment reproducibility.
- **Experiment Tracking:** Integrated `WandBConfig` provides comprehensive experiment tracking with real-time metric logging, hyperparameter sweeps, and model artifact management.
- **Type Safety and Validation:** Added precise type annotations throughout the codebase, enabling `mypy` enforcement and Pydantic validation for all configuration objects, catching potential runtime errors at development time.

Further relevant components can be found in the directory tree in subsection A.5.

# 4 Data Methodology within UniTraj

This section presents the data processing methodology within the UniTraj framework [7], covering the transformation from heterogeneous autonomous driving datasets to standardized tensor representations suitable for trajectory prediction models. UniTraj serves as a unified interface that harmonizes datasets from WOMD [22], Argoverse2 [5], and nuScenes [4] through the ScenarioNet format [21].

## 4.1 UniTraj's Data Processing Pipeline

The UniTraj preprocessing pipeline transforms raw, heterogeneous data into standardized, model-ready tensors through a multi-phase process. Each phase is implemented as a distinct method within the `DataParser` class, ensuring modularity and clarity. The following provides a brief qualitative overview of the pipeline; for detailed implementations and more comprehensive descriptions, see subsection A.4.

**Phase 1: Temporal Window Extraction** Extracts fixed-length time windows from raw trajectories and applies frequency masking for uniform temporal sampling (2).

**Phase 2: Map Feature Processing** Converts raw map data (lanes, boundaries) into standardized polylines with uniform point density and semantic encodings (3).

**Phase 3: Agent Selection** Filters for relevant agents based on motion and observation quality to identify suitable prediction candidates (4).

**Phase 4: Coordinate Transformation** Transforms the entire scene into an agent-centric coordinate frame for each candidate, ensuring translation and rotation invariance (5).

**Phase 5: Feature Assembly** Constructs comprehensive feature vectors for each agent, concatenating spatial, kinematic, and semantic attributes (6).

**Phase 6: Proximity Filtering & Padding** Selects the $N_{\max}$ closest agents and pads the agent tensor to a fixed size for batching (7).

**Phase 7: Map Tensorization** Segments, resamples, and selects the $K_{\max}$ closest map polylines, creating a fixed-size map tensor (8).

**Phase 8: Future Processing** Processes and transforms the ground truth future trajectories for the center agent (9).

**Phase 9: DatasetItem Assembly** Assembles all processed tensors and masks into a final `DatasetItem`, the fundamental unit of the dataset (10).

**The `BaseDataParser` Class.** The `BaseDataParser` serves as the central orchestrator of the data processing pipeline. It coordinates the parallel processing of scenario chunks, manages HDF5 file storage of the resulting dataset, and assembles the metadata that is collected throughout the pipeline.

## 4.2 The UniTraj DatasetItem and Batching

The output of the processing pipeline is a collection of `DatasetItem` instances, each encapsulating a complete prediction scenario in a standardized format.

**The `DatasetItem` Structure.** A `DatasetItem` is a strongly typed object that holds all the `numpy` arrays corresponding to a single, agent-centric view of a scene. This includes the historical agent states, the map geometry, ground truth future trajectories, and associated validity masks. The `DatasetItem` contains all necessary data for training, evaluation, analysis and visualization, and hence contains many redundant features.

**Batching with the `collate_fn`.** For model training and inference, individual `DatasetItem`s are grouped into batches by a PyTorch `DataLoader` within a `pl.DataModule`. This process is orchestrated by a custom `collate_fn`, which transforms a list of `DatasetItem`s into a typed dictionary of tensors which can subsequently be used as model input.

The key tensors within this batch structure are described below. For complete tensor specifications, dimensions, and feature breakdowns, refer to Tables 4, 7, and 8 in subsection A.1.

**Dynamic Agent Representation.** Agent trajectories are encoded in tensor $\boldsymbol{X}_d \in \mathbb{R}^{B \times N_{\max} \times T_p \times F_{ap}}$, where $B$ is the batch size. It contains comprehensive state information for up to $N_{\max}$ agents over $T_p$ historical timesteps. The agent feature dimension $F_{ap}$ encompasses spatial coordinates, physical dimensions, one-hot encoded object types, temporal position embeddings, heading, and kinematic states. The corresponding validity mask $\boldsymbol{M}_d \in \{0,1\}^{B \times N_{\max} \times T_p}$ indicates data availability.

**Static Map Representation.** High-definition map information is represented through tensor $\boldsymbol{X}_s \in \mathbb{R}^{B \times K_{\max} \times L \times F_{map}}$, encoding up to $K_{\max}$ polylines with $L$ points each. The $F_{map}$ features capture geometric and semantic properties, including polyline point coordinates, direction vectors, and lane type classifications. The validity mask $\boldsymbol{M}_s \in \{0,1\}^{B \times K_{\max} \times L}$ handles variable map complexity.

**Ground Truth and Auxiliary Data.** Future trajectory targets for the center agent are provided in $\boldsymbol{y}_c \in \mathbb{R}^{B \times T_f \times F_{af}}$, where $F_{af}$ includes position and velocity over $T_f$ prediction timesteps. The batch also contains metadata like Kalman-difficulty scores and classifications of the type of maneuver that the center agent is performing.

**Sample Metadata and Dataset Composition.** UniTraj maintains metadata for efficient dataset management and analysis. The sample metadata, stored as a pandas DataFrame that provides high-level information about each scenaro. Each entry is uniquely identified by a composite index encoding the dataset name, worker process, scenario counter, and agent index (e.g., `av2_scenarionet-0-0-0`). A complete overview of the sample metadata fields is provided in Table 9.

**DatasetItem Visualization.** To illustrate the comprehensive nature of the processed data, Figure 5 presents a detailed visualization of a representative `DatasetItem`. This enhanced visualization demonstrates the multi-modal nature of the data, showing the spatial relationships between agents, their historical trajectories, and the underlying high-definition map infrastructure.

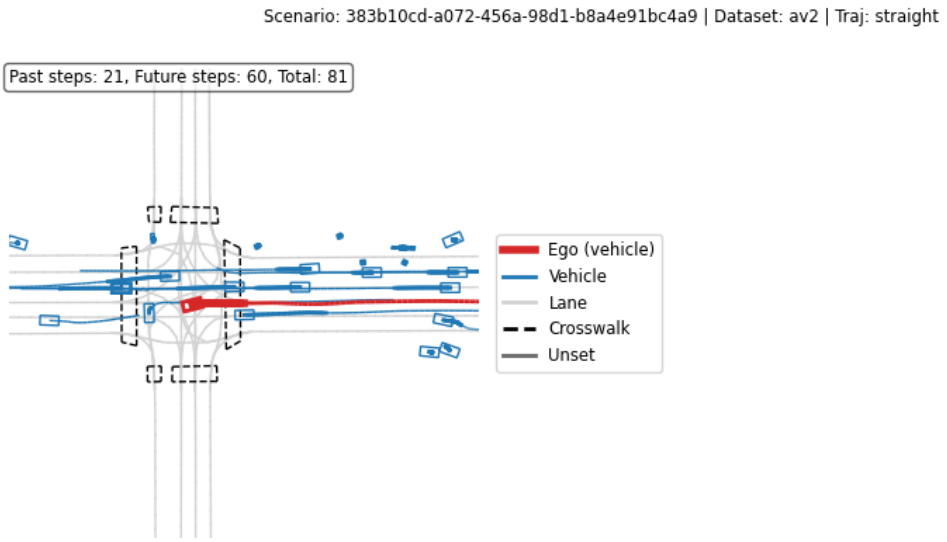



Figure 5: **Enhanced DatasetItem Visualization.** A comprehensive view of a processed scenario showing: (a) the center agent (ego vehicle) in red with its historical trajectory, (b) surrounding agents with their temporal information including velocity vectors and heading indicators, (c) high-definition map polylines with semantic classifications (lanes in blue, crosswalks in black dashed, boundaries in gray), and (d) agent interaction zones and proximity relationships. This visualization demonstrates the multi-modal nature of the standardized `DatasetItem` format, capturing spatial and temporal dynamics used for trajectory prediction tasks.

# 5 Model Architecture and Functional Decomposition

## 5.1 CASPNet: Rasterized BEV Encoding with Dual FPN

This section distills the key ideas behind *Context-Aware Scene Prediction Network* (CASPNet) [9] and its transformer successor *CASPFormer* [15]. Both architectures perform *joint multi-agent* trajectory forecasting from rasterized bird's-eye-view (BEV) inputs, yet they differ strongly in how they fuse context and decode trajectories. We summarize their core components, strengths, and limitations.

**CASPNet** [9] processes rasterized BEV inputs through a dual-encoder architecture that separately handles dynamic agent trajectories and static HD map information. Overall, it strongly resembles a *fully convolutional network* as it employs no fully connected layers, meaning that the spatial representation of the scene is preserved throughout the network. Furthermore, it can be classified as a *feature pyramid network* (FPN) [23] as it provides the decoder with latent representations at multiple spatial resolutions. Together, these architectural qualities starkly resemble the famous *U-Net* architecture with *lateral skip connections* [24]. Considering CASPNet's usage of *attention* mechanisms within the skip connections its closest relative in the field of image segmentation is the *Attention U-Net* [25].

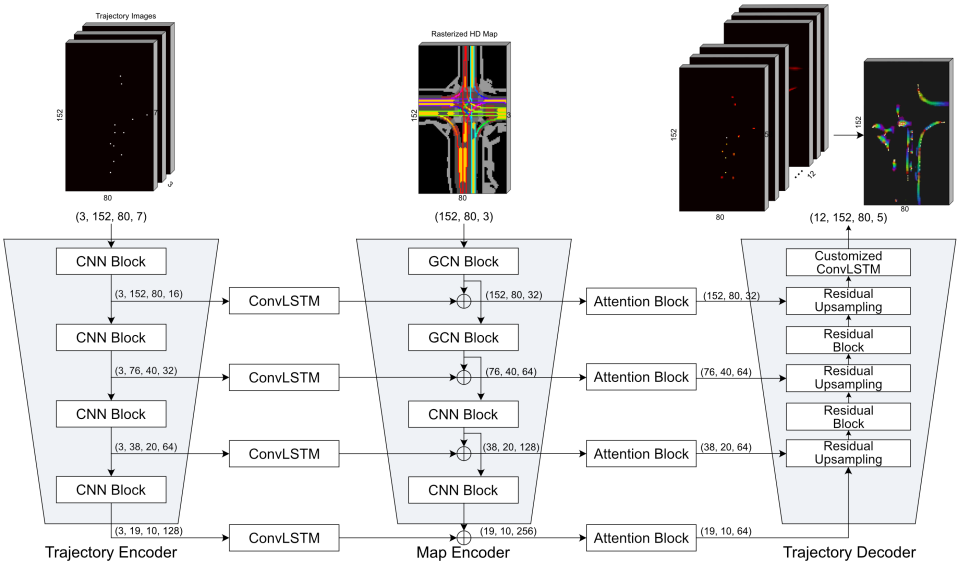



Figure 6: CASPNet architecture overview: dual FPN encoders with Gabor filters, pixel-adaptive attention, and grid-based ConvLSTM decoder [9].

The CASPNet architecture is illustrated in Figure 6 and consists of the

following components:

**Dual FPN encoders.** Separate feature-pyramid networks process

(i) a BEV stack of past agent trajectories $\mathbf{I}_d \in \mathbb{R}^{T_p \times H \times W \times F_d}$

(ii) a static HD-map raster $\mathbf{I}_s \in \mathbb{R}^{H \times W \times F_s}$

At each level $\ell$, the trajectory branch produces:

$$\mathbf{F}_\ell^{\text{traj}}(t) = \text{CNN}_\ell\big(\mathbf{I}_d(t)\big) \in \mathbb{R}^{T_p \times C_\ell \times H_\ell \times W_\ell} \tag{5}$$

where all $T_p$ BEV frames representing the agent trajectories are encoded independently by the same CNN, yielding a *time-dependent multi-scale feature map* $F_\ell^{\text{traj}}(t) \in \mathbb{R}^{C_\ell \times H_\ell \times W_\ell}$ at each pyramid level $\ell$ and timestep $t \in \{0, \dots, T_p - 1\}$.

The static map branch employs *steerable Gabor filters*[2] in the first two convolutional blocks. These filters are particularly well-suited for capturing orientation and scale-sensitive features like road networks, where lane markings exhibit strong orientation and scale-specific patterns. A Gabor filter combines a Gaussian envelope with a sinusoidal carrier wave, allowing it to simultaneously localize features in space and frequency. This makes it highly effective for detecting elongated and parallel structures across varying scales and rotations [26].
The map features are computed as follows:

$$\mathbf{F}_\ell^{\text{map}} = \text{CNN}_\ell^{\circ}(\mathbf{I}_s) \in \mathbb{R}^{C_\ell \times H_\ell \times W_\ell}, \tag{6}$$

**Temporal fusion.** ConvLSTM cells at every pyramid level aggregate the temporal context across all $T_p$ timesteps yielding a single feature map per level [27]:

$$\mathbf{F}_\ell^{\text{traj}} = \text{ConvLSTM}_\ell\Big\{\mathbf{F}_d^\ell(t)\Big\}_{t=1}^{T_p} \in \mathbb{R}^{C_\ell \times H_\ell \times W_\ell}, \tag{7}$$

which are then concatenated with the static map features at each level:

$$\boldsymbol{\mathcal{F}} = \{\mathbf{F}_\ell^{\text{traj}} \oplus \mathbf{F}_\ell^{\text{map}}\}_{\ell=0}^{L_{\text{FPN}}-1}, \quad \text{where } \mathbf{F}_\ell \in \mathbb{R}^{2C_\ell \times H_\ell \times W_\ell} \tag{8}$$

$\boldsymbol{\mathcal{F}}$ is the resulting latent feature pyramid.

**Pixel-adaptive attention.** The *Attention Block* in the lateral skip connections (Figure 6) adaptively blends the receptive fields of multiple *dilated convolutions* [28] per spatial location, capturing multi-scale interactions through learned per-pixel attention weights over different dilation rates,

[2] See subsection A.3 for further details.

similar to [25]. For each location $(i, j)$ at pyramid level $\ell$, the mechanism computes:

$$\mathbf{F}_{\ell}^{(i,j)} = \sum_{r} \alpha_r^{(i,j)} \cdot \text{DilatedConv}_r(\mathbf{F}_{\ell}^{(i,j)}), \tag{9}$$

where $\alpha_r^{(i,j)} = \text{softmax}(\text{Conv}(\mathbf{F}_{\ell}^{(i,j)}))$ are learned attention weights that dynamically select appropriate receptive field sizes for each spatial location.

**Grid-based decoder.** The decoder employs a series of *residual up-sampling blocks* to progressively up-sample the feature maps from the FPN to the original raster resolution. Each residual block consists of a parallel *transposed convolution* and a *linear interpolation* layer. The resulting feature maps of each up-sampling block are concatenated with the corresponding feature maps the next level of the FPN.
Finally, the decoder uses a *ConvLSTM* layer to autoregressively generate future predictions. For each future timestep $t \in \{T_p, \ldots, T_p + T_f - 1\}$, the decoder outputs a occupancy probabilities and motion offsets. The occupancy probabilities are represented as a categorical distribution over the possible agent classes at each pixel, while the motion offsets represent the displacement of each pixel with respect to the previous timestep [9].

**Pros**

- ⊕ Inference time independent of agent count
- ⊕ Inherent multi-modality via heat-map superposition
- ⊕ Good interpretability through adaptation of well-established motifs from the field of image segmentation

**Cons**

- ⊖ Metric accuracy capped by raster resolution
- ⊖ Inference cost scales with grid size $\mathcal{O}(H \cdot W)$
- ⊖ Vectorized outputs require post-processing
- ⊖ No *explicit* prediction of $M$ multi-modal
- ⊖ Doesn't respect identities of agents
- ⊖ Limited temporal modeling as the time dimension is squeezed
- ⊖ No explicit relationship modeling between agents or between agents and map

- ⊖ No symmetries in the agent-centric coordinate system for all but the ego agent
- ⊖ Only suited for *single-agent* motion forecasting

The follow-up work CASPNet++ [14] improves upon CASPNet by replacing its single heat-map head with a two-stage design to capture spatio-temporal occupancy grids for every actor, as well as an Agent decoder that transforms the grid cells of selected targets into explicit, multi-modal trajectory splines, thereby modelling interactions more richly and introducing the ability to predict $M$ explicit multi-modal trajectories. However, while CASPNet++ improves upon the original architecture in terms of accuracy and interaction modeling and allows for *true* joint multi-agent forecasts, we will not discuss it in detail.

## 5.2 CASPFormer: Hybrid CNN-Transformer with Deformable Attention

**CASPFormer** [15] retains CASPNet's, FPN-CNN backbone while replacing its grid-based decoder with a transformer that emits vectorized $(x, y)$ trajectories, explicitly modeling the uncertainties of each mode as a *parametrized Laplacian*. The core innovation lies in adapting *deformable attention* [29] from object detection to trajectory prediction, enabling sparse, efficient attention over multi-scale feature representations. The overall architecture is illustrated in Figure 7, with the detailed decoder mechanism shown in Figure 8. The following section requires basic understanding of both regular and deformable attention mechanisms. While we are referring the reader to [30] and [29] for a detailed introduction, a concise explanation of the key concepts is provided in subsection A.2.

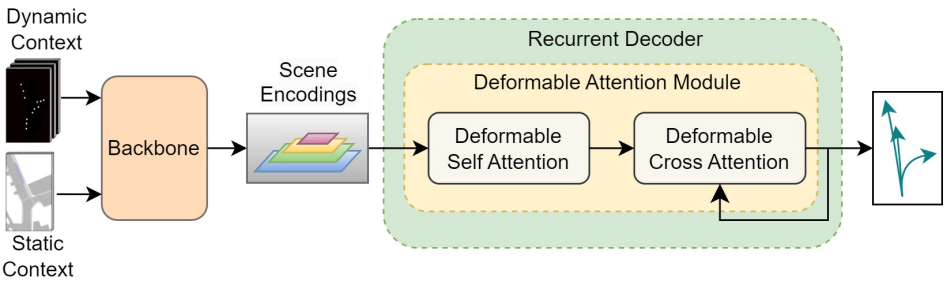



Figure 7: CASPFormer overall architecture: CNN+FPN backbone processes rasterized BEV inputs, deformable self-attention fuses multi-scale features, and recurrent deformable cross-attention autoregressively decodes vectorized trajectories [15].

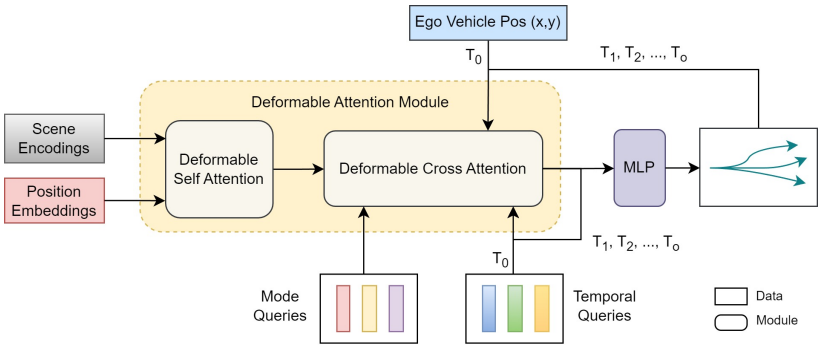



Figure 8: CASPFormer decoder architecture with temporal and mode queries, deformable cross-attention, and autoregressive trajectory generation [15].

The CASPFormer as depicted in Figure 8 decoder consists of the following key components:

**Multi-Scale Deformable Self-Attention (MSDA).** The FPN feature maps $\boldsymbol{\mathcal{F}} = \{\mathbf{F}_\ell\}_{\ell=0}^{L-1}$ as previously defined in Equation 8 are first flattened and enriched with non-learnable 2D sinusoidal positional encodings as introduced in [30] to preserve spatial relationships across different scales. MSDA performs feature fusion by allowing each spatial location to attend to relevant features across all pyramid levels:

$$\mathbf{Z}^{\mathrm{SA}} = \sum_{n=1}^{N_H} \mathbf{W}_n \left[ \sum_{\ell=0}^{L_{\mathrm{FPN}}-1} \sum_{k=1}^{K_s} A_{n\ell qk} \cdot \mathbf{F}_\ell(\phi_\ell(\mathbf{p}_q) + \boldsymbol{\Delta p}_{n\ell qk}) \right] \tag{10}$$

Here, *queries* and *keys* are derived from the same flattened feature maps $\mathrm{Flatten}(\boldsymbol{\mathcal{F}} + \mathbf{PE})$. The learned offsets $\boldsymbol{\Delta p}_{\ell nkq} \in \mathbb{R}^2$ enable each query to attend to informative locations across the entire feature pyramid, while being constrained to all features of the n-th attention head. $\phi_\ell$ re-scales the normalized coordinates $\mathbf{p}_q \in [0,1]^2$ to the spatial dimensions of the feature map at level $\ell$.
The output of the MSDA is a latent representation of the entire scene, encoding all spatio-temproal and social dynamics of the agents and the map. Hence, this module should be seen as part of the scene-encoder, rather than the decoder.

**Recurrent Deformable Cross-Attention.** The fused features $\mathbf{Z}^{\mathrm{SA}}$ serve as *keys* and *values* for the cross-attention mechanism. CASPFormer employs a dual-query approach that combines *temporal queries* and *mode queries* to capture both temporal dependencies and facilitate mode diversity. The query construction is defined as:

$$\begin{aligned}\mathbf{Q}_t^{\text{temp}} &= \mathbb{1}(t = T_p) \cdot \mathbf{E}_{\text{init}} + \mathbb{1}(t > T_p) \cdot \mathbf{Z}_{t-1}^{\text{CA}} \\ \mathbf{Q}^{mode} &\in \mathbb{R}^{M \times d} \quad \text{(learnable mode embeddings)} \\ \mathbf{Q}_t &= \mathbf{Q}_t^{temp} \oplus \mathbf{Q}^{mode} \quad \text{(concatenated dual queries)}\end{aligned} \tag{11}$$

The temporal queries maintain temporal coherence across timesteps, while mode queries provide fixed embeddings to promote behavioral diversity and mitigate mode collapse.

For each timestep $t$, the deformable cross-attention decoder updates the reference point $\mathbf{p}^{ref}$ as the last position of the ego-vehicle $\hat{\boldsymbol{\mu}}_{t-1}$ based on previous predictions and computes:

$$\mathbf{Z}_t^{\text{CA}} = \sum_{m=1}^{M} \mathbf{W}_m \left[ \sum_{k=1}^{K_s} A_{nkq} \cdot \mathbf{Z}^{\text{CA}}(\mathbf{p}^{ref} + \boldsymbol{\Delta p}_{nkq}) \right] \tag{12}$$

where both the attention weights and offsets are computed as delineated in subsection A.2. The DCA block depicted in Figure 8 stacks three deformable cross-attention layers, each with its own set of learnable parameters, to refine the predictions iteratively. This allows the model to capture complex interactions and dependencies across multiple timesteps and modes.

**Autoregressive Decoding with Mixture Outputs.** The outputs of the final cross-attention layer are processed by an MLP to generate trajectory distributions as a Laplacian mixture, which empirical work has shown to match the heavy-tailed statistics of real human motion better than Gaussian mixtures [18, 15]:

$$\begin{aligned}\hat{\boldsymbol{\mu}}_t, \hat{\mathbf{b}}_t &= \text{MLP}(\mathbf{Z}_t^{\text{CA}}) \\ \hat{\boldsymbol{\pi}} &= \text{softmax}(\text{MLP}(\mathbf{Z}_t^{\text{CA}}))\end{aligned} \tag{13}$$

where $\hat{\boldsymbol{\mu}}_t \in \mathbb{R}^{N_c \times M \times 2}$ are Cartesian position means, $\hat{\mathbf{b}}_t \in \mathbb{R}^{N_c \times M \times 2}$ are uncertainty scales, and $\hat{\boldsymbol{\pi}}_t \in \mathbb{R}^{N_c \times M}$ are mode probabilities for each of the $M$ trajectory modes and each agent of interest; $N_c$ being the number of agents in the scene, whose trajectories are to be predicted.

The recurrent architecture updates both the temporal queries and reference point at each timestep:

$$\mathbf{p}_{(t+1)}^{ref} = \text{argmax}_m \hat{\boldsymbol{\pi}}_{t,ego} \cdot \hat{\boldsymbol{\mu}}_{t,m^{ego}} \tag{14}$$

where $m^* = \text{argmax}_m \hat{\boldsymbol{\pi}}_{t,ego}$ yields the index of the ego agent's most probable mode, hence the next reference point is set the likeliest predicted position of the ego vehicle (which must be one of the $N_c$ agents of interest).

### Loss Formulation and Training Recipe

CASPFormer adopts the HiVT loss formulation [18], combining regression and classification objectives $\mathcal{L} = \mathcal{L}_{reg} + \lambda \mathcal{L}_{cls}$ to optimize the predicted mixture of Laplacians. To avoid numerical instability and mode collapse in mixture model optimization [31], the regression loss adopts winner-takes-all (WTA) strategy, where only the mode with the lowest Euclidean distance to the ground truth trajectory is considered:

$$\mathcal{L}_{reg} = -\frac{1}{T_f} \sum_{t=1}^{T_f} \log[\mathbb{L}(P_t \mid \mu_t, b_t)] \tag{15}$$

The influence of all other modes and the predicted categorical distribution is captured by a cross-entropy classification loss:

$$\mathcal{L}_{cls} = -\frac{1}{M} \sum_{k=1}^{M} \log(\pi_k) \mathbb{L}(P_{T_f,k} \mid \mu_{T_f,k}, b_{T_f,k}) \tag{16}$$

**Pros**

- ⊕ Direct vectorized output without post-processing
- ⊕ Explicit multi-modal prediction and uncertainty estimates
- ⊕ Deformable attention enables efficient Transformer decoding
- ⊕ Using the past-predictions as temporal queries introduces grounding and interpretability
- ⊕ Mode-queries mitigate mode collapse

**Cons**

- ⊖ Still inherits quantization artifacts from the rasterized backbone
- ⊖ Limited to ego-agent-centric coordinate systems
- ⊖ BEV raster discretization artifacts in crowded intersections
- ⊖ Self-attention encoder scales quadratically with the spatial dimensions of the feature pyramid, i.e. $\mathcal{O}(H^2 \cdot W^2)$
- ⊖ Cross-attention decoder complexity scales as $\mathcal{O}(N_c HW)$

## 5.3 Motion TRansformer (MTR)

### Architectural Structure

The Motion TRansformer (MTR) framework provides a unified approach to multimodal motion prediction, a critical task for autonomous systems [10]. Its central principle is to model motion prediction as a joint optimization of two tasks: **global intention localization** and **local movement refinement** [10]. In simpler terms, the model first identifies a diverse set of high-level goals or destinations for an agent and then fine-tunes the precise paths to reach those goals.

The architecture consists of two main stages:

1. **Scene Context Encoding:** An encoder processes historical data from all agents and map features to build a rich understanding of the scene, including interactions between elements.
2. **Prediction Generation:** A decoder uses a special set of "Motion Queries" to propose and refine multiple future trajectories for an agent of interest based on the encoded scene context.

This modular, Transformer-based structure allows MTR to effectively manage the inherent uncertainty and multimodality of traffic scenarios.

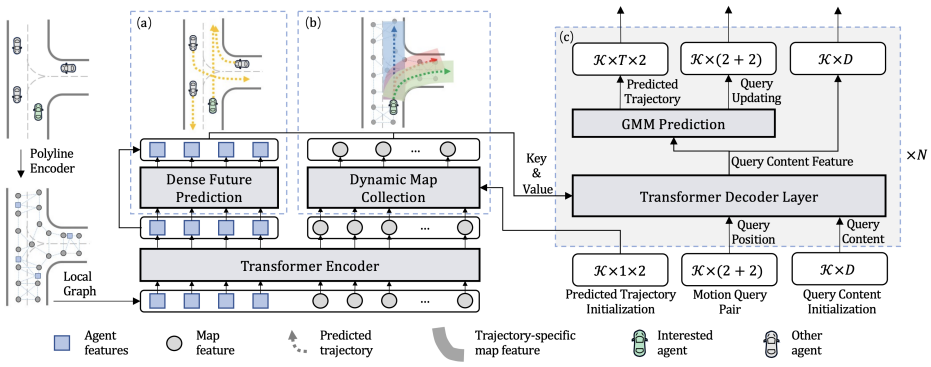



Figure 9: A high-level diagram of the MTR architecture, illustrating the flow from vectorized inputs through the scene encoder and motion decoder to the final multimodal trajectory outputs. (Based on Figure 1 from [10]).

### Input Encoding

MTR represents all input data—both from agents and the map—as vectorized polylines, which are sequences of points. This method, pioneered by models like VectorNet, is more efficient and precise than grid-based representations like bird's-eye-view (BEV) images, as it avoids quantization errors and

handles sparse map data well [11]. PointNet-like encoders then process these polylines to generate fixed-size feature vectors for the Transformer model [32].

**Map Encoders**
Map features are represented as a collection of polylines, where each polyline describes a map element like a lane centerline, road boundary, or crosswalk. Each point within a polyline has attributes such as its position (x, y, z) and direction vector. A five-layer MLP processes the points of each map polyline, and a max-pooling operation aggregates this information into a single feature vector ($M_p$) for that polyline [10]. This vector is then projected to a 256-dimensional space to match the agent feature dimensionality.

**Agent Encoders**
An agent's historical state is also treated as a polyline. The input for each agent includes its motion history (position, size, heading, velocity) over the last second. To help the model understand the data, several specific features are included:

- **One-hot Category Mask:** This vector identifies the agent's type (e.g., Vehicle, Pedestrian, Cyclist). The model is designed to handle these different agent classes.
- **One-hot Time Embedding:** This feature explicitly encodes the timestep of each point in the history. This is crucial for the Transformer architecture, which does not otherwise have a built-in sense of sequence order [33].

A three-layer MLP processes these point-wise features, and max-pooling creates a 256-dimensional feature vector ($A_p$) for each agent's history.

### Trajectory Decoding

Once the inputs are encoded, the core of the MTR model performs context aggregation and generates predictions.

**Transformer Encoder Module.**
The feature vectors for all agents ($A_p$) and map polylines ($M_p$) are fed into a Transformer Encoder. This module's job is to contextualize each element by modeling its interactions with its surroundings. To do this efficiently, MTR uses a **local self-attention** mechanism. Instead of having each element attend to every other element in the scene, it only attends to its $k$ nearest neighbors (e.g., $k = 16$). This preserves local structure, reduces computational cost, and improves performance compared to global attention [10]. The output of this stage is a set of context-aware feature vectors for agents ($A_{past}$) and the map ($M$).

**Dynamic Map Collection Strategy**

To ensure that trajectory refinement is guided by the most relevant parts of the map, MTR uses a Dynamic Map Collection strategy within its decoder. For each stage of trajectory prediction, the model identifies the $L$ map polylines (e.g., $L = 128$) that are spatially closest to the current trajectory being generated. These selected map features are then used in the decoder's cross-attention mechanism, allowing the model to focus on the immediate local geometry (like lane boundaries for a lane-change maneuver) to make precise, context-aware adjustments [10].

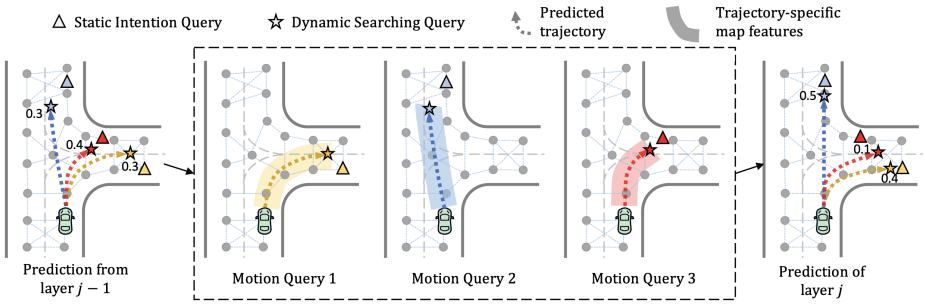



Figure 10: An illustration of the Dynamic Map Collection strategy. As a trajectory is refined (dashed line), the model selects the L closest map polylines (highlighted) to inform the next prediction step. (Based on Figure 3 from [10]).

**Motion Decoder Module (Transformer Decoder Layer)**

The Motion Decoder generates the final trajectory predictions. Its key innovation is the use of **Motion Query Pairs** to guide this process [10]. This query-based approach is conceptually derived from the object query paradigm pioneered by DETR in object detection [34], but is adapted here for trajectory forecasting. Each of the $K$ pairs (e.g., $K = 64$) consists of a static query and a dynamic query.

- **Intention-Driven Querying:** The process is driven by two types of queries:
  - **Static Intention Queries ($Q_I$):** These queries act as stable anchors for different motion modes (e.g., turn left, go straight, stop). They are derived from a set of representative "intention points", which are pre-calculated by clustering the endpoints of ground-truth trajectories from the training data. Each static query specializes in a specific high-level intention, which helps disentangle the prediction of diverse behaviors and stabilizes training [10].
  - **Dynamic Searching Queries ($Q_S^j$):** These queries perform the fine-

grained refinement. At each decoder layer, the dynamic query is updated based on the trajectory predicted by the previous layer. It then probes the scene context (both agent and map features) to gather the specific local details needed to refine that trajectory.

- **Goal-Conditioned Prediction:** The static queries ($Q_I$) enable goal-conditioned prediction. By grounding each of the $K$ prediction modes in a fixed, data-driven intention point, the model generates trajectories that are explicitly conditioned on achieving these latent goals. This synthesizes the strengths of goal-based methods (which are good at capturing multi-modality [13]) and direct-regression methods (which are good at generating refined paths [35]).

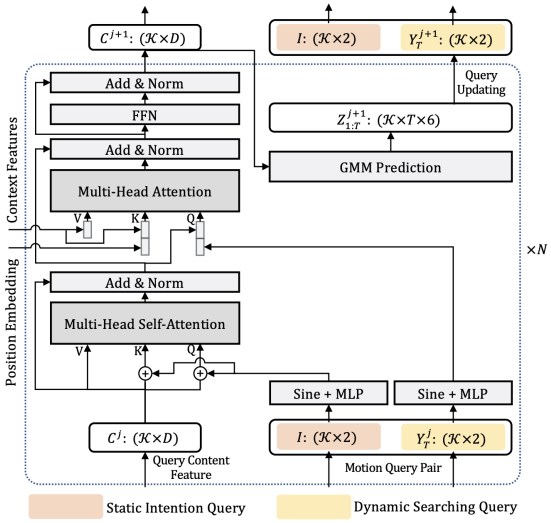



Figure 11: The architecture of a single Motion Decoder layer, showing how the static intention query ($Q_I$) and dynamic searching query ($Q_S^j$) interact with scene context to refine a trajectory hypothesis. (Based on Figure 2 from [10]).

### Probabilistic Multimodal Output Generation

To represent its multimodal predictions, MTR uses a Gaussian Mixture Model (GMM) as its final output layer. This approach is a direct application of Mixture Density Networks [36], which combine a neural network with a mixture model to approximate arbitrary conditional probability distributions. The features from the final decoder layer are passed through an MLP (prediction head) to generate the parameters for $K$ different Gaussian distributions. Each of these $K$ components represents one possible future mode and includes:

- A mean trajectory (a sequence of waypoints).

- A probability score ($p_k$), which reflects the model's confidence in that specific mode.

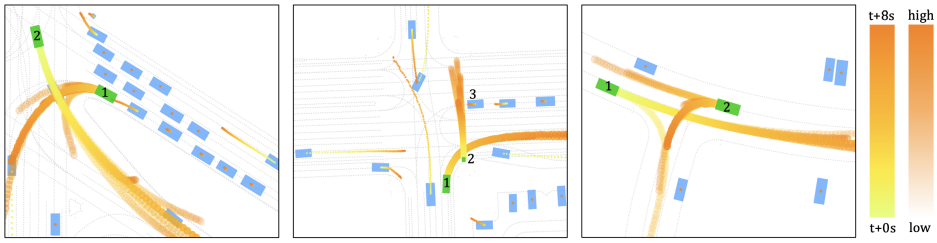
This probabilistic output provides a rich distribution over plausible futures rather than a single deterministic guess.

**Iterative Query Refinement and Training**
The decoder consists of multiple stacked layers (e.g., 6 layers). This stacked structure enables an iterative refinement process. The output trajectory from layer $j-1$ is used to update the dynamic searching query ($Q_S^j$) and to guide the Dynamic Map Collection for layer $j$. Thus, an initial rough trajectory from the first layer is progressively corrected and detailed by subsequent layers, which can focus on more relevant local information [10].

The model is trained by comparing its $K$ predictions to the single ground-truth trajectory. A loss function guides the model to improve its predictions over time. It essentially encourages the model to assign a high probability ($p_k$) to the predicted trajectory that most closely matches the ground truth, while also moving the waypoints of that trajectory even closer to the actual path. This process can be thought of as a learned form of hypothesis testing: the queries propose $K$ hypotheses, and the training process teaches the model how to evaluate and refine them based on scene context [10].

## Input-Output Formulation

At a high level, the MTR model transforms historical scene data into a set of ranked, probable future paths.




(a) **V2** is passing the intersection to turn left with high speed. Our model predicts multimodal behaviors for **V1:** turn left or make a U-turn. In any case, **V1** is predicted to yield for **V2**.

(b) **P2** is passing the road through the crosswalk while **V1** is on the right-turn lane to turn right. Both **V1** and **V3** are predicted to yield for **P2**.

(c) Our model predicts multimodal behaviors for **V1**: go straight and turn right, since it still has a distance to the intersection. **V2** is predicted to yield for **V1** when turning left, since **V1** is moving fast towards the intersection.

Figure 12: Qualitative examples of MTR's multimodal predictions in interactive scenarios. The model generates multiple future trajectories (orange paths) for an agent of interest, with agent shwon in green. Other agents are represented by blue boxes. The examples highlight the model's ability to predict socially compliant behaviors, such as yielding to another vehicle (a), a pedestrian (b), or a fast-moving car at an intersubsection (c).

- **Input Domain:** The minimum necessary input for a single agent's prediction is its historical spatiotemporal data and the features of the surrounding map environment. All input coordinates are normalized into an agent-centric frame, which makes the learning task simpler and invariant to global position and rotation. The framework is designed to process data from large-scale datasets like the Waymo Open Motion Dataset (WOMD) [22]. The model can handle various agent types, including vehicles, pedestrians, and cyclists. Within a scene, MTR processes up to 128 context agents, from which a smaller group of up to 8 "agents of interest" are designated for prediction, as specified by the WOMD benchmark [22]. The provided documentation focuses on the MTR architecture itself and does not detail specific dataset fusion techniques (e.g., for UniTraj) or agent selection criteria (e.g., based on Kalman difficulty).
- **Output Domain:** The model's exact output is a set of $K$ plausible future trajectories (e.g., $K = 64$) for an agent of interest over a future time horizon (e.g., 8 seconds). Each trajectory is defined by a sequence of waypoints and is associated with a probability score ($p_k$) indicating its likelihood. For evaluation against benchmarks that require a smaller number of predictions (e.g., 6), a Non-Maximum Suppression (NMS) algorithm is used to select the most likely and diverse trajectories from the initial set of $K$ proposals [10].

**Illustrative Scenario: Agent at a Crosswalk**
To understand how the components work together, consider a vehicle approaching a crosswalk where a pedestrian is present.

1. **Input and Encoding:** The model takes in the vehicle's past trajectory and the pedestrian's history, along with map polylines for the crosswalk and stop line. These are encoded into feature vectors.
2. **Contextualization:** The Transformer Encoder's local attention mechanism allows the vehicle's feature vector to be influenced by the nearby crosswalk and pedestrian features, creating a context-aware representation. An auxiliary task predicts the likely future motion of the pedestrian, further enriching the scene context [10].
3. **Decoding and Prediction:** The Motion Decoder begins its work. Static intention queries propose several plausible high-level actions, such as "yield before the crosswalk" and "proceed through".
   - The "yield" hypothesis is refined. Its dynamic query focuses attention on the stop line (via Dynamic Map Collection) and the pedestrian's state.
   - The "proceed" hypothesis is refined similarly.

4. **Final Output:** After iterative refinement across the decoder layers, the GMM output will reflect the context. Because the model has taken the pedestrian's presence and future path into account, the probability ($p_k$) associated with the "yield" trajectory will be high, while the "proceed" trajectory will have a low probability. This demonstrates how MTR generates contextually-aware, multimodal predictions. A visualization of this would show the vehicle's past path, the map features (lanes, crosswalk), and multiple predicted future paths, each with a different color and a displayed probability score.

### Key Scientific Challenges and Model Variants

The primary scientific challenge in motion prediction is managing the inherent uncertainty and multimodality of agent behavior in complex, interactive environments. MTR addresses this through its query-based decoder, but different versions of the model optimize for different goals.

- **MTR (Standard):** The baseline model uses $K = 64$ queries to generate a rich set of proposals, which are then filtered using NMS. This approach excels at capturing a wide variety of possible behaviors, leading to high performance on metrics like mean Average Precision (mAP) [10].
- **MTR-e2e:** This "end-to-end" variant uses only 6 motion queries and no NMS. It is designed for scenarios where a fixed, small number of predictions is required. It uses a different training strategy better suited to the small, adaptive query set [10].
- **MTR++:** This extension tackles the more complex problem of simultaneous multi-agent prediction. It introduces a shared "Symmetric Scene Context" encoder and allows the intention queries of different agents to interact via "Mutually-Guided Intention Querying" [37]. This enables the model to produce more efficient and scene-compliant joint predictions for all agents at once.

These variants show the flexibility of the core MTR architecture in addressing the diverse challenges of motion forecasting.

### Pros

⊕ Vectorized polyline input and output avoid rasterization artifacts and preserve geometric precision.

⊕ Query-based decoder enables explicit, interpretable multi-modal prediction.

⊕ Dynamic map collection ensures trajectory refinement leverages the most pertinent map features at each step.

⊕ Probabilistic output via GMM provides uncertainty estimates and supports downstream risk assessment.

**Cons**

⊖ Computational cost scales with the number of queries and decoder layers.

⊖ NMS post-processing is required to select diverse, high-quality predictions for benchmarks with limited output slots.

⊖ Static intention queries rely on clustering endpoints, which may not capture rare or atypical maneuvers.

⊖ No explicit modeling of physical feasibility constraints or agent kinematics.

## 5.4 LMFormer: Lane based Motion Prediction Transformer

LMFormer [2] is a fully query-centric, transformer-based architecture for joint multi-agent trajectory forecasting. It ingests both static lane segments and dynamic motion vectors in their local frames, embeds them via learnable Fourier features, and processes them through self- and cross-attention blocks before iteratively decoding multiple modes for each agent. The high-level architecture is shown in Figure 13.

### Encoder

The encoder leverages the asymmetric relationship between static and dynamic contexts: while static lane geometry is independent of dynamic agent states, agent behavior is heavily influenced by road structure. This motivates a two-stage design where static context is encoded first, then incorporated into dynamic agent processing. Following the original architecture [2], all scalar inputs are embedded using learnable Fourier features before processing through specialized attention modules.

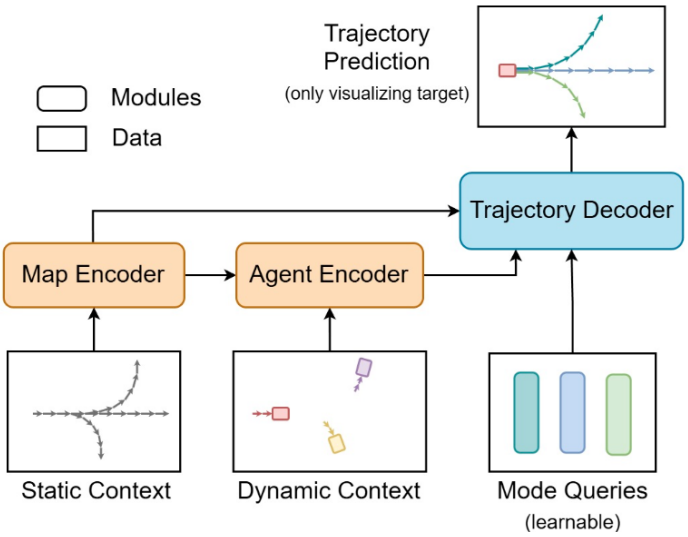



Figure 13: LMFormer architecture: transformer encoders for static and dynamic contexts & recurrent cross-attention decoder with coarse-to-fine refinement [2].

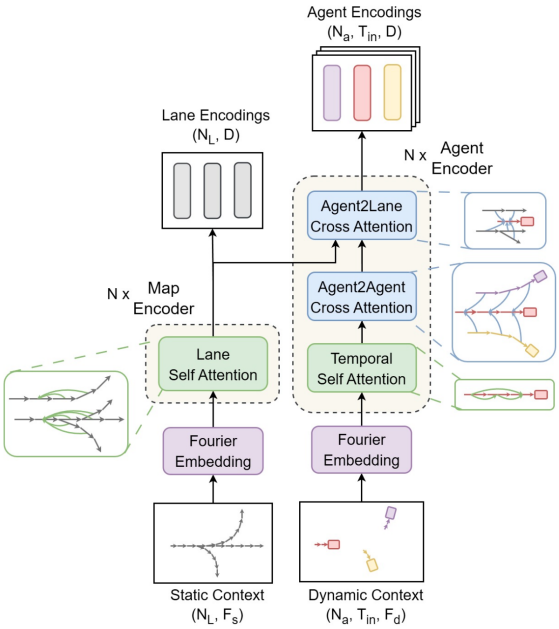



Figure 14: LMFormer encoder: learnable Fourier embeddings, lane self-attention, agent temporal and cross-attentions [2].

**Learnable Fourier Embeddings**
All scalar inputs—lane-segment lengths, relative headings, time offsets, motion-vector norms, and relative descriptors—are lifted with *learnable Fourier features* [20]:

$$\mathbf{x} \mapsto \mathrm{GeLU}\left(\mathbf{W}_A \left[\sin(2\pi \mathbf{x}\mathbf{W}_f^T) \oplus \cos(2\pi \mathbf{x}\mathbf{W}_f^T)\right] + \mathbf{B}_\varphi\right) + \mathbf{B}_\psi \tag{17}$$

where all $\mathbf{W}_\bullet$ and $\mathbf{B}_\bullet$ are learnable parameters:

- $\mathbf{W}_f$ encodes which harmonic frequencies are emphasized for each scalar input dimension,
- $\mathbf{W}_A$ represents the amplitude of each frequency in the harmonic basis,
- $\mathbf{B}_\varphi$ are learnable phase shifts for each harmonic, and $\mathbf{B}_\psi$ is a learnable bias of the final embedding.

This harmonic basis lets later attention layers pick up patterns at various task-adaptive spatio-temporal frequencies. Unlike fixed sinusoidal embeddings, the learnable frequencies in $\mathbf{W}_f$ adapt during training to capture the most informative scales for trajectory prediction—from fine-grained local maneuvers to coarse lane-following behaviors.

- **Map (Lane) Encoder:** Applies $N_{es}$ layers of multi-headed self-attention over $N_L = K \cdot L$ lane-segment tokens to capture topological and geometric relations between lane segments. Each segment is represented by its query-centric Fourier embeddings, with relative position embeddings incorporated into the keys/values to maintain spatial relationships. The resulting *static scene encodings* are $\mathbf{E}_s \in \mathbb{R}^{N_L \times D}$.
- **Agent Encoder:** Stacks three attention modules over $N$ agent tokens with $T_p$ timesteps:
  1. Temporal self-attention models the temporal dependencies of each agent, allowing each agent to attend to its own past motion history.
  2. Agent-agent cross-attention to model social interactions between agents per timestep, enabling agents to attend to the encodings of other agents at the same timestep.
  3. Agent-lane cross-attention to the $N_L \times D$ dimensional lane encodings (keys + values) and dynamic agent encodings (queries).

  The encoder repeats this triad $N_{ed}$ times, producing *agent encodings* $\mathbf{E}_d \in \mathbb{R}^{N \times T_p \times D}$.

## Recurrent Cross-Attention Decoder

To facilitate multimodal trajectory prediction, LMFormer introduces learnable mode queries where each query corresponds to a distinct trajectory for

every agent. The decoder employs a two-level strategy combining iterative refinement across $N_{\text{dec}}$ stacked layers with autoregressive generation within each layer. Inspired by DAB-DETR's coarse-to-fine anchor refinement [38], early layers establish coarse trajectory patterns while later layers add fine-grained details, maintaining temporal coherence and social consistency.

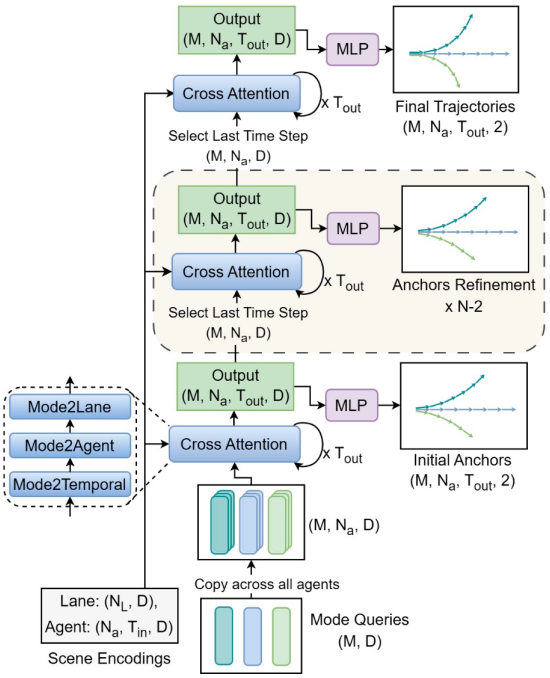



Figure 15: [2] LMFormer decoder: recurrent mode-query cross-attention modules that iteratively generate future motion vectors.

The architecture incorporates recurrent temporal decoding ($\times T_f$) within cross-attention modules, where each recurrent loop updates both query position and content, similar to CASPFormer [15]. Only the final timestep representations from each block are propagated forward, creating a bottleneck that forces the model to compress learned temporal dynamics into a compact representation. Each motif employs three cross-attention modules to capture various interactions:

- **Mode2Temporal Cross-Attention:** Aggregates temporal information of agent encodings (keys/values) along the time-axis to mode queries, aligning each behavioral hypothesis with the agent's observed motion history.
- **Mode2Agent Cross-Attention:** Models social interactions where each mode of an agent attends to the same mode of other agents at the same historic timestep, ensuring multi-modal futures are grounded in historic

social context.

- **Mode2Lane Cross-Attention:** Grounds trajectory modes in static road geometry by letting mode queries attend to lane segment encodings, incorporating drivable directions, topology constraints, and map alignment.

Each decoder block $i \in \{1, \dots, N_{\text{dec}}\}$ maintains a query tensor $\mathbf{Q}^{(i)} \in \mathbb{R}^{M \times N \times T_f \times D}$ where $M$ is the number of modes per agent.

**Output Formulation.** Following [2], every mode query ultimately predicts a *motion-vector chain*

$$\mathcal{T}_{out}^{a,m} = \left[(V_1^{a,m}, S_1^{a,m}), \dots, (V_{T_f}^{a,m}, S_{T_f}^{a,m})\right], \tag{18}$$

where $V_t^{a,m} = [P_{t-1}^{a,m}, P_t^{a,m}]$ is a displacement vector and $S_t^{a,m} \in \mathbb{R}^2$ its uncertainty. Each $(V_t, S_t)$ pair parameterises one component of a *Laplacian mixture* density. To elucidate the workings within the decoder, we present its simplified algorithmic description in 1.

**Algorithm 1** LMFormer Recurrent Cross-Attention Decoder

**Input:** Agent encodings $\mathbf{E}_d \in \mathbb{R}^{N \times T_p \times D}$, Lane encodings $\mathbf{E}_s \in \mathbb{R}^{N_L \times D}$, Learned Mode anchors $\mathbf{A} \in \mathbb{R}^{M \times D}$
**Output:** Trajectory sets $\{\mathcal{T}_{out}^{(i)}\}_{i=1}^{N_{\text{dec}}}$

$\mathbf{q} \leftarrow \text{repeat}(\mathbf{A}, N)$ ▷ Initialize mode queries $(M, N, D)$
**for** $i = 1$ **to** $N_{\text{dec}}$ **do**
  $\mathbf{Q}_{\text{seq}}^{(i)} \leftarrow \text{zeros}(M, N, T_f, D)$
  **for** $t = 1$ **to** $T_f$ **do**
    ▷ **Mode2Temporal**
    $\mathbf{q} \leftarrow \text{CrossAttn}(\mathbf{q}, K = V = \mathbf{E}_d[\text{same } n, \tau = 1 \dots, T_p, :])$
    ▷ **Mode2Agent**
    $\mathbf{q} \leftarrow \text{CrossAttn}(\mathbf{q}, K = V = \mathbf{E}_d[\tilde{n} = 1 \dots n, \text{same } \tau, :])$
    ▷ **Mode2Lane**
    $\mathbf{q} \leftarrow \text{CrossAttn}(\mathbf{q}, K = V = \mathbf{E}_s)$
    $\mathbf{Q}_{\text{seq}}^{(i)}[:, :, t, :] \leftarrow \mathbf{q}$
  **end for**
  $\mathcal{T}_{out}^{(i)} \leftarrow \text{MLP}(\mathbf{Q}_{\text{seq}}^{(i)})$ ▷ $(M, N, T_f, 4)$
**end for**

**Loss formulation**

LMFormer re-uses winner-takes-all regression loss from Equation 15, combined with the cross-entropy loss to update mode probabilities from Equation 16 but supervises *every* refinement layer (except the first) to encourage iterative coarse-to-fine anchor refinement:

$$\mathcal{L} = \lambda \mathcal{L}_{cls} + \sum_{n=2}^{N_{\text{dec}}} \mathcal{L}_{reg}^{(n)}, \tag{19}$$

where $\lambda$ balances classification against regression [2].

## Discussion

### Static Context Limitations

Contrary to the claim in [2] that connections between segments with similar heading should dominate, the learnable Fourier encoding should be capable of representing much more complex relations between segments. As Eq. (17) suggests, the learnable Fourier+MLP can capture relations beyond mere heading similarity. The authors used this hypothesis to justify not reaching SOTA performance on minFDE@1 on nuScenes, claiming that the model might fail to forecast trajectories not aligned with lane segments.

Lane graphs are intrinsically sparse, replacing dense attention with deformable or other sparse attention schemas would reduce this block's cost from $\mathcal{O}(N_L^2)$ to $\mathcal{O}(N_L K_s)$ without harming accuracy, mirroring the gains CASPFormer achieved in the raster domain.
Their ablation studies showed that long-range dependencies in lane encodings do not appear significant, which supports our claim, that encoding mechanisms that focus on local relationships and sparsely on long-range relationships might be beneficial. [8] employed graph neural networks to model the lane graph, which would be a potential alternative to the self-attention mechanism employed in LMFormer's map encoder. We question why the authors only encoded lane segments while other static scene elements listed in Table 8 were not included in the static scene encoder, as they obviously carry important information for trajectory forecasting. However, we acknowledge that lanes play a dominant role in conditioning feasible trajectories during decoding, so it might be worthwhile to decode the static scene context in a way that allows the resulting latents to be decomposed into lane segments and other static scene elements.

As [2] notes, future work could explore encoding explicit physical feasibility constraints (e.g., speed limits) inside the decoder.

**Offset-based Refinement** LMFormer predicts a *full*, absolute trajectory at every decoder stage. A direct analogue to DAB-DETR would instead predict *local offsets* $\Delta_t^{(n)}$ and update $\hat{P}_t^{(n)} = \hat{P}_t^{(n-1)} + \Delta_t^{(n)}$, keeping each refinement inside a bounded error ball and improving gradient conditioning. Employing a residual formulation is generally seen as a measure to improve gradient stability, allow for faster convergence and smoother paths [38]. While [2] notes that the autoregressive nature necessary for motion forecasting conflicts with the offset-based approach, we cannot understand why this

would be the case.

**Velocity-Position Decoupling and Optimization** As [2] notes, it would be beneficial to redict velocity mixtures in a separate head. Velocities are usually provided as ground-truth and could hence be optimized directly, furthermore integrating them would yield a second branch to predict trajectory positions, grounding these predictions in the velocity profile and contributing to physical continuity.

While [2] weighs the regression losses across all refinement layers uniformly, we believe that later layers should dominate the regression loss. This is because early layers primarily establish coarse trajectory patterns, while later layers refine these patterns into fine-grained, multi-modal trajectories. Furthermore, the gradient emitted from the loss of the later layers will also affect all previous layers.

**Geometric Symmetries & Multi-Agent Capabilities** The architecture preserves all symmetries as introduced in item 2.2. These properties allow the same cached scene encoding to serve *all* agents and *all* successive frames, a capability absent in all agent-centric architectures.

**Generalization & Data Diversity** The authors remarked on limited generalization capabilities when cross-testing on other datasets. This is exacerbated by the lack of diverse training data and unbalanced data distributions in nuScenes, where some maneuvers occur less frequently. UniTraj already provides a classification of maneuver types (which could be extended) such that those that are underrepresented could be oversampled during training to improve generalization capabilities. This supports our remarks in section 4 on why a unified dataset via a framework like UniTraj will be necessary to achieve improved generalization capabilities.

The authors furthermore noted the need for targeted data augmentation strategies to improve robustness and mode diversity; such augmentation strategies would be a natural extension of the UniTraj framework, as these augmentations could benefit all sorts of trajectory forecasting models, not just LMFormer.

**Pros**

- ⊕ Fully query-centric, preserving $\mathrm{SE}(2) \rtimes \mathbb{R}$ invariance end-to-end.
- ⊕ Joint multi-agent decoding with shared static context.
- ⊕ Recurrent refinement yields temporally coherent, multi-modal trajectories.

⊕ Managed to drastically reduce VRAM requirements compared to [8], as they trained on a single A100 GPU with 40GB VRAM, while QCNet required approx. 160GB VRAM for training.

**Cons**

⊖ Higher VRAM demand compared to CASPFormer due to larger key-value cache.

⊖ Limited static context representation (lane-only tokens).

⊖ Uniform loss weighting across refinement layers rather than emphasizing later stages.

⊖ Generalization challenges across different datasets due to training data limitations.

⊖ Employed only a single type of augmentation (flipping inputs along the driving direction).

**Relation to CASPNet & CASPFormer**
CASPNet operates on raster grids and predicts per-pixel occupancies; CASPFormer adds deformable attention and vector outputs but retains an *agent-centric* frame. LMFormer discards the raster backbone entirely, embraces the query-centric paradigm, and gains strict symmetry compliance and parallel multi-agent decoding—at the cost of a larger key-value cache and higher VRAM demand (≈2x CASPFormer on nuScenes).

# 6 Experimental Design and Performance Evaluation

This chapter details the framework for training and evaluating the trajectory prediction model. It outlines the experimental setup, the learning objectives, and the metrics used for assessment, followed by a quantitative and qualitative analysis of the model's performance.

## 6.1 Training and Evaluation Paradigm

The training and evaluation environment uses a robust and reproducible setup. This subsection details the computational environment, configuration management, and the data processing pipeline.

### Training Environment and Configuration

The entire suite for training, evaluation, and logging is managed through PyTorch Lightning, with experiment tracking and visualization handled by Weights & Biases (WandB). This setup facilitates managed logging, checkpointing, and distributed training strategies.

All configurations and hyperparameters are managed by Pydantic. This enables a "Config-as-Factory" pattern, which simplifies the process of swapping different models, datasets, or loss functions. While the environment is robustly configured, the provided documentation does not detail specific hyperparameters such as batch size or learning rate.

### Data Handling and Processing

For high-throughput data ingestion, the data loading pipeline uses HDF5. The framework enhances processing efficiency by randomly partitioning the full sample index into 32 shards and assigning one shard to each DataLoader worker, which ensures balanced and parallel prefetching.

The training data undergoes a detailed processing pipeline before being fed to the model:

- **Agent Selection:** First, the pipeline selects agents—vehicles, pedestrians, or cyclists—based on their movement distance and visibility within scenarios.
- **Scenario Classification:** Scenarios receive a difficulty classification (Easy, Medium, Hard) derived from a Kalman filter analysis.

- **Coordinate Normalization:** The system then normalizes all coordinate data into an agent-centric frame using the rotation matrix $R_z(-\theta_c)$.
- **Tensor Assembly:** Finally, it assembles agent and map features into padded and masked tensors, specifically $\mathbb{R}^{N_{max} \times T_p \times F_{ap}}$ for dynamic agent data and $\mathbb{R}^{K \times L \times F_{map}}$ for static map data, preparing them for model ingestion.

## 6.2 Optimization Strategy and Learning Objective

The MTR model was trained from scratch without the use of a pre-trained model. A significant portion of the implementation involved adapting the model to the UniTraj framework's data parsing and configuration systems. The primary learning objective is to train the model to accurately forecast multimodal trajectories.

The UniTraj framework unifies loss functions for different models, and the overall training objective is to minimize these functions to improve prediction accuracy. The training process specifically seeks to decrease metrics like the minimum Final Displacement Error (FDE). A key component of the objective is the Brier Final Displacement Error, which assesses both the accuracy of the trajectory's final point and the model's confidence in its prediction. This objective guides the model to learn multimodal distributions. However, the exact mathematical formulation of the loss function used for this training run is not specified in the source material. Similarly, the specific optimization algorithm, such as Adam or SGD, and its associated parameters like learning rate and scheduling, are not mentioned.

## 6.3 Evaluation Metrics

The model's performance assessment relies on a set of standard industry metrics. The mathematical definitions for these metrics are as follows:

- **Average Displacement Error (ADE):** The mean L2 distance between the predicted trajectory and the ground truth across all time steps, defined as $ADE = \mathbb{E}_t[||\hat{y}_t - y_t||_2]$.
- **Final Displacement Error (FDE):** The L2 distance between the final predicted position and the ground truth final position, given by $FDE = ||\hat{y}_T - y_T||_2$.
- **Miss Rate (MR):** The fraction of predictions where the FDE for the most likely trajectory exceeds a distance threshold $d_{thresh}$ (e.g., 2.0 m). The formula is $MR = \mathbb{E}_k[\mathbb{I}\{||\hat{y}_T^{(k)} - y_T||_2 > d_{thresh}\}]$.
- **Brier Final Displacement Error (BrierFDE):** A metric that scores both trajectory accuracy and its assigned probability, defined as $BrierFDE =$

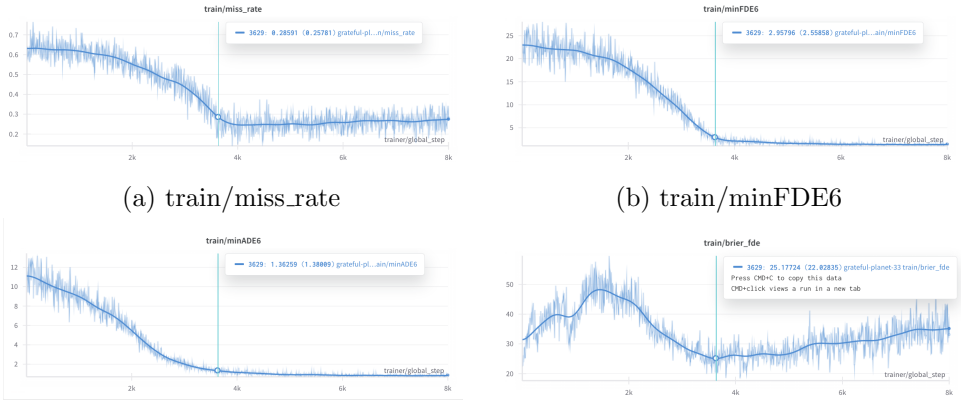
$\mathbb{E}_k[p_k \cdot ||\hat{y}_T^{(k)} - y_T||_2^2]$.

## 6.4 Performance Analysis

This subsection evaluates the trained MTR model's performance through quantitative metrics and qualitative review. The analysis confirms the model's learning effectiveness and its ability to generate accurate and probable trajectory forecasts.

### Quantitative Performance Analysis

The quantitative analysis of the MTR model's performance is based on metrics recorded during training and validation. The training process shows a clear reduction in loss values, indicating the model is learning from the data. The plots in Figure 16 exhibit a characteristic learning curve: a steep reduction in error during initial training, which then gradually plateaus. This trend indicates convergence.




(a) train/miss_rate (b) train/minFDE6

(c) train/minADE6 (d) train/brier_fde

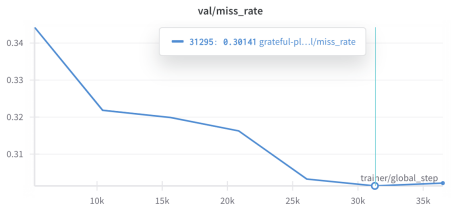
Figure 16: A grid of performance metrics on the training set. The plots show a consistent decrease in error for miss rate, brier FDE, minFDE6, and minADE6 over approximately 6,000 training step of batches, which indicates successful learning.

Final validation metrics, taken at step 31,295 (6,259 batches), are presented in Table 1. On the validation set, the trained MTR model achieved a brier-minFDE of 1.98, a minFDE of 1.6655, a minADE of 0.86294, and a Miss Rate of 0.30141. These results demonstrate competitive performance against baseline values from existing literature. The consistent decrease in error seen in Figure 17 demonstrates that the model generalizes well to unseen data. The model shows improvements in brier-minFDE (1.98 vs 2.08) and

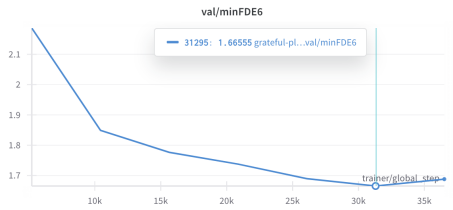
minFDE (1.6655 vs 1.68) compared to the UniTraj paper benchmarks, while maintaining comparable performance in minADE and Miss Rate [7].

Table 1: Final Validation Metrics at Step 31,295

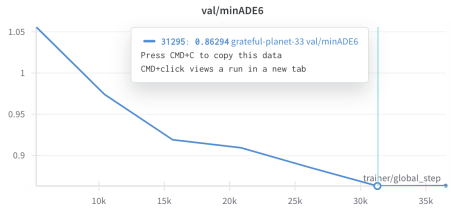
| Metric | Value | Benchmark (MTR Paper) |
|---|---|---|
| brier-minFDE | 1.98 | 2.08 |
| minFDE | 1.6655 | 1.68 |
| minADE | 0.86294 | 0.85 |
| Miss Rate | 0.30141 | 0.30 |



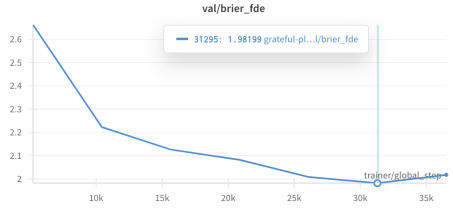

(a) val/miss_rate: 0.30141




(b) val/minFDE6: 1.6655




(c) val/minADE6: 0.86294




(d) val/brier_fde: 1.98199

Figure 17: Performance metrics on the validation set during MTR training. The plots show a consistent decrease in error for miss rate, FDE, ADE, and Brier-FDE as training progresses.

### Qualitative Analysis with Sample Case Illustration

Figure 18 shows a representative output from the MTR model in a right-turn scenario. The model predicts several future trajectories for the agent, each assigned a probability. The green line marks the ground truth path, while colored lines indicate the predicted modes. The most probable prediction (p=0.36) aligns closely with the actual trajectory, while a less likely mode (p=0.02) reflects an alternative, plausible maneuver.

This example demonstrates key aspects of MTR's architecture in practice. The vectorized polyline representation preserves geometric precision without

rasterization artifacts. The query-based decoder produces explicit, interpretable multi-modal predictions with clear confidence scores. The dynamic map collection mechanism ensures trajectory refinement leverages relevant map features, enabling context-aware predictions that reflect the road geometry and driving constraints. The sample case highlights both strengths and limitations of the MTR architecture. The model produces high-precision trajectory outputs and assigns explicit probabilities to each mode, supporting risk-aware decision making in autonomous systems. However, the computational requirements scale with query count and decoder depth, which can impact real-time performance. The need for non-maximum suppression to select diverse predictions introduces additional post-processing overhead.

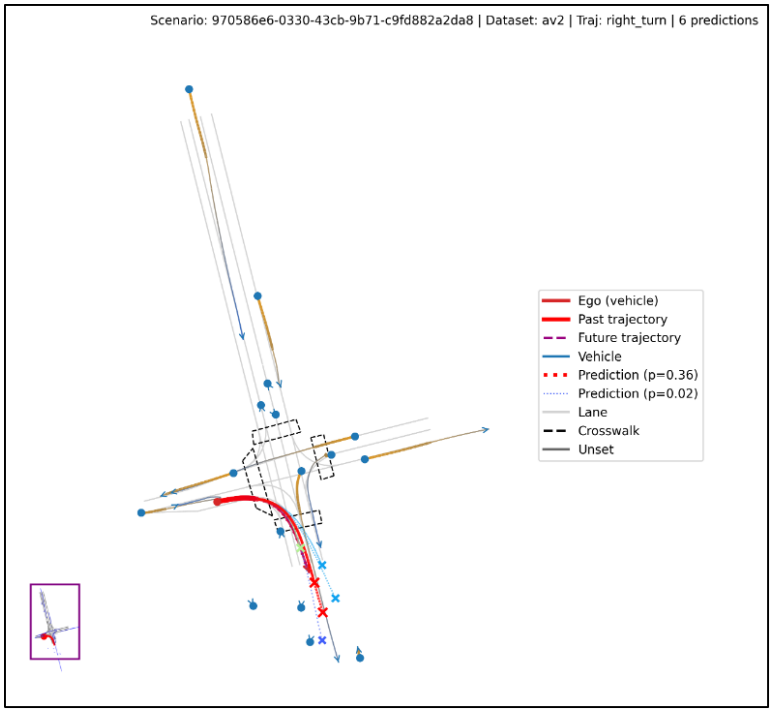



Figure 18: A multimodal prediction from the MTR model for a right-turn scenario. The green line is the ground truth, and the colored lines are predictions with different probabilities.

# 7 Conclusion and Discussion

This seminar work provided a comprehensive theoretical analysis of selected trajectory prediction methods in autonomous driving. The investigation examined multiple approaches, from the raster-based CASPNet to the transformer-based CASPFormer, MTR and LMFormer architectures,

elucidating key design principles and interesting architectural features. In our section 2, we provided a detailed qualitative comparison of commonly used scene representation paradigms, paying particular attention to the differences between agent-centric and query-centric approachs, examining their respective strengths and weaknesses. This analysis highlighted how query-centric methods enable permutation and SE(2)-invariance while supporting streaming inference and joint prediction, contrasting with the limitations of agent-centric frameworks bound to privileged reference frames.

Our practical work resulted in significant improvements to the UniTraj framework, refactoring it in accordance with the principles of modern deep learning and fixing various issues. We established a robust experimental pipeline using PyTorch Lightning and achieved a brier-minFDE of 1.98 compared to the MTR benchmark of 2.08.

## 7.1 Reflection on Initial Objectives

The initial aim of this seminar work was to implement a minimal, simplified model for *joint multi-agent* trajectory prediction. However, the complexities of the chosen task became apparent incrementally throughout the project. The inherent intricacies of the existing algorithms, and working with a framework like UniTraj, as well as limited computational constraints necessitated a significant scope reduction to focus on a comparative analysis between different algorithms instead of implementing a "Very-Smol-Single-Agent-Motion-Forecasting" algorithm.

The complexities within the field of multi-modal trajectory prediction are presented throughout this report, particularly the comparative study in Chapter 2 examining agent-centric versus query-centric paradigms and the detailed description of the different models in Chapter 5, demonstrates the depth of the field, as well as the comprehensiveness of our investigation. While the original implementation goal proved unattainable, the work achieved a thorough understanding of various important concepts and design choices, whose implications reach far beyond the domain of motion forecasting.

Engaging with state-of-the-art transformer architectures deepened our understanding of attention mechanisms for encoding spatial and temporal relationships. The work explored how cross attention fuses different modalities and examined design choices such as mode queries and anchors. Applying a Geometric Deep Learning perspective to the query-centric paradigm fostered understanding of concepts including isomorphisms and fiber bundles. The experimental work demonstrated practical application of these concepts.

This seminar work succeeded in building conceptual foundations and developing a critical perspective on current research directions. The experience highlighted the importance of clear abstractions and robust data handling

practices while exposing the inherent challenges of adapting complex research frameworks during this seminar.

Ultimately, our ambitious choice of problem revealed itself as disproportionately complex for a single-semester module, creating considerable pressure in the closing weeks. Acknowledging this mis-scoping and persevering through the stress it caused has become a formative lesson in realistic project framing that complements the countless theoretical insights we gained.

# A Appendix

## A.1 Notation and Symbol Reference

This section provides a comprehensive reference for all symbols and conventions used throughout this work, following UniTraj framework standards [7] and the DatasetItem type definitions. The following tables have been created using [39], given the type and shape definitions in unitraj/datasets/-types.py.

Table 2: Temporal dimension symbols and definitions

| Symbol | Definition |
|---|---|
| $T_p, T_{\text{in}}$ | Number of past timesteps (historical context) |
| $T_f, T_{\text{out}}$ | Number of future timesteps (prediction horizon) |
| $T$ | Total trajectory length: $T = T_p + T_f$ |
| $\Delta t$ | Temporal sampling interval (i.e. 0.1s for 10Hz data) |

Table 3: Spatial and agent dimension symbols

| Symbol | Definition |
|---|---|
| $N_{\text{max}}$ | Maximum number of agents in scene (i.e. 64) |
| $N, N_A$ | Actual number of agents in a specific scenario |
| $N_c$ | Number of center agents, $N_c = N$ in [2]. |
| $K_{\text{max}}$ | Maximum number of map polylines (i.e. 256) |
| $K$ | Actual number of map polylines in a specific scenario |
| $L$ | Number of points per map polyline (i.e. 20) |
| $N_L$ | Number of lane segments in $\boldsymbol{X}_s, N_L = K \cdot L$ |
| $F_{ap}$ | Agent feature dimension (i.e. 39) |
| $F_{af}$ | Agent future state dimension (4: $x, y, v_x, v_y$) |
| $F_{map}$ | Map feature dimension (i.e. 29) |
| $D$ | Hidden dimension of encodings or latent representations. |

Table 4: Primary data tensors and their shapes

| Tensor | Shape and Description |
|---|---|
| $\boldsymbol{X}_d$ | $[N_{\max}, T_p, F_{ap}]$ Agent trajectory features |
| $\boldsymbol{M}_d$ | $[N_{\max}, T_p]$ Agent trajectory validity mask |
| $\boldsymbol{X}_{d,pos}$ | $[N_{\max}, T_p, 3]$ Agent positions $(x, y, z)$ |
| $\boldsymbol{X}_{d,last}$ | $[N_{\max}, 3]$ Last observed agent positions |
| $\tilde{\boldsymbol{X}}_d$ | $[N_{\max}, T_f, F_{af}]$ Agent future states |
| $\tilde{\boldsymbol{M}}_d$ | $[N_{\max}, T_f]$ Agent future validity mask |
| $\boldsymbol{y}_c$ | $[T_f, F_{af}]$ Center agent ground truth trajectory |
| $\boldsymbol{M}_c$ | $[T_f]$ Center agent ground truth validity mask |
| $\boldsymbol{X}_s$ | $[K_{\max}, L, F_{map}]$ Map polyline features |
| $\boldsymbol{M}_s$ | $[K_{\max}, L]$ Map polyline validity mask |
| $\boldsymbol{C}_s$ | $[K_{\max}, 3]$ Map polyline centers |

Here, $\tilde{\bullet}_\circ$ denotes tensors, expressing future states, while $\bullet_\circ$ denotes tensors for historical or static context. $\bullet_d$ denotes all tensors, representing *dynamic* agents, while $\bullet_s$ denotes tensors for *static* map elements. The center agent, whose trajectory is the target for prediction, is denoted by the subscript $c$.

Table 5: Agent feature components ($F_{ap} = 39$)

| Component | Indices | Description |
|---|---|---|
| Spatial | [0:6] | Position $(x, y, z)$, bbox-dimensions $(l, w, h)$ |
| Type | [6:11] | One-hot agent type encoding as per 7 |
| Temporal | [11:33] | One-hot time embedding ($T_p + 1$ dimensions) |
| Heading | [33:35] | Heading encoding $(\sin\theta, \cos\theta)$ |
| Kinematic | [35:39] | Velocity $(v_x, v_y)$, acceleration $(a_x, a_y)$ |

The utilized type encodings are reflecting the respective entity types, which are provided by ScenarioNet [21], which uses MetaDrive [40] internally. It should be noted that the use of these type encodings comes at the loss of more fine-grained semantic annotations that are provided by the original Argoverse2 dataset.

Table 6: Map feature components ($F_{map} = 29$)

| **Component** | **Indices** | **Description** |
|---|---|---|
| Position | [0:3] | Current point coordinates $(x, y, z)$ |
| Direction | [3:6] | Direction vector $(d_x, d_y, d_z)$ |
| Previous | [6:9] | Previous point coordinates $(x_{prev}, y_{prev}, z_{prev})$ |
| Lane Type | [9:29] | One-hot lane type encoding (20 categories) |

Table 7: Agent type encodings

| **Type** | **ID** | **Description** |
|---|---|---|
| UNSET | 0 | Unknown or unclassified agent |
| VEHICLE | 1 | Cars, trucks, buses, motorcycles |
| PEDESTRIAN | 2 | Pedestrians, wheelchair users |
| CYCLIST | 3 | Bicyclists, e-scooter riders |

The PolylineType enumeration defines the mapping from MetaDrive/ScenarioNet polyline types to integer labels used in the $F_{\text{map}}$ feature encoding. This enumeration is derived from the MetaDrive simulation environment [40] and ScenarioNet dataset format [21].

Table 8: PolylineType Enumeration as per MetaDrive

| Integer | Description |
|---|---|
| 0 | Default/unspecified polyline type |
| 1 | Highway or freeway lane |
| 2 | Surface street lane |
| 3 | Dedicated bicycle lane |
| 6 | Broken single white line |
| 7 | Solid single white line |
| 8 | Solid double white line |
| 9 | Broken single yellow line |
| 10 | Broken double yellow line |
| 11 | Solid single yellow line |
| 12 | Solid double yellow line |
| 13 | Passing zone double yellow line |
| 15 | Road boundary line |
| 16 | Median boundary |
| 17 | Stop sign location |
| 18 | Pedestrian crosswalk |
| 19 | Speed bump or traffic calming |

Note that the conversion from AV2 → ScenarioNet → UniTraj causes many categories to collapse, and plenty of the categories as per Table 8 and Table 7 are actually used, and hence result in degenerate encodings.

### Dataset Metadata

## A.2 Deformable Attention

Traditional self and cross-attention in transformers [30] scales quadratically with spatial resolution $\mathcal{O}(H^2W^2)$[3], making it computationally infeasible for high-resolution feature maps. Deformable attention [29] addresses this by attending to a sparse set of sampling locations $\{\mathbf{p}_q + \boldsymbol{\Delta p}_{nqk}\}_{k=1}^{K_s}$ around each query position $\mathbf{p}_q$, where $\boldsymbol{\Delta p}_{nqk}$ are dynamically computed *fractional* offsets and $K_s$ is the number of sampling points per head. This reduces the complexity to $\mathcal{O}(NHK_s)$[4].

For a query with content features $\mathbf{Z}_q$ with reference point $\mathbf{p}_q \in [0,1]^2$ and an input feature map $\mathbf{X}$ multi-head deformable attention with $N_h$ heads

[3] Note that the sequence consists of $H \times W$ image patches or activations, and not the original image resolution.

[4] Simplified for single head attention

Table 9: UniTraj Sample Metadata Fields

| Field | Description | Data Type |
|---|---|---|
| `h5_path` | Path to the HDF5 file containing the processed sample data | Path |
| `scenario_id` | Original scenario identifier from ScenarioNet | str |
| `kalman_difficulty` | Array of Kalman filter difficulty scores for all agents in the scenario | np.ndarray |
| `num_agents` | Total number of agents with valid trajectories in the scenario | int64 |
| `num_agents_interest` | Number of agents of interest (prediction candidates) in the scenario | int64 |
| `scenario_future_duration` | Number of future timesteps available in the scenario | int64 |
| `num_map_polylines` | Total number of map polylines in the scenario | int64 |
| `track_index_to_predict` | Index of the specific agent track to predict within the scenario | int64 |
| `center_objects_type` | Semantic type of the centered object (vehicle, pedestrian, cyclist) | category |
| `dataset_name` | Name of the source dataset (e.g., av2_scenarionet) | category |
| `trajectory_type` | Behavioral classification of the trajectory | category |

computes:

$$\text{DeformAttn}(\mathbf{Z}_q, \mathbf{X}, \mathbf{p}_q) = \sum_{n=1}^{N_H} \mathbf{W}_n \left[ \sum_{k=1}^{K_s} A_{nkq} \cdot \mathbf{X}(\mathbf{p}_q + \boldsymbol{\Delta p}_{nkq}) \right] \tag{20}$$

where $\mathbf{W}_m$ are learned projection matrices and $\mathbf{X}(\bullet)$ represents bilinear sampling from the feature map $\mathbf{X} \in \mathbb{R}^{C \times H \times W}$. Both offsets $\boldsymbol{\Delta p}_{nkq}$ and attention weights $A_{nkq}$ are computed via linear projections of the query and key features:

$$\tilde{\bullet} \propto \text{activ}(\underbrace{\mathbf{Z}_q^T \mathbf{U}_n^T}_{\mathbf{Q}} \underbrace{\mathbf{V}_n \mathbf{X}_k}_{\mathbf{K}}) \quad \text{with activ} = \begin{cases} \text{softmax for } \tilde{\bullet} = A_{nkq} \\ \text{sigmoid for } \tilde{\bullet} = \boldsymbol{\Delta p}_{nkq} \end{cases} \tag{21}$$

$\mathbf{U}_n$ and $\mathbf{V}_n$ are learned projection matrices to compute the queries $\mathbf{Q}$ and *keys* $\mathbf{K}$ in the $n$-th attention head. In case of *self*-attention, all the queries, keys, and values are derived from the same sequence, i.e. $\mathbf{Z}_q = \mathbf{X}$, while in case of *cross*-attention, the queries stem from a different sequence than the keys and values. In a simpler single-head attention setting the keys are computed as $\mathbf{K} = \mathbf{XW}$.

**Conceptual Interpretation.** To understand deformable attention, it helps to conceptualize the key components [39]:

**Queries (Q).** Represent *what information is being sought.* In trajectory prediction, queries encode the current prediction state (temporal context and mode embeddings) that determines what spatial features to attend to for generating the next trajectory point.

**Keys (K) and Values (V).** Keys act as *indices* that determine the compatibility between queries and spatial locations, while values contain the *actual feature content* to be aggregated. In CASPFormer, both are derived from the multi-scale CNN feature maps, encoding spatial scene context at different resolutions.

**Reference Points ($\mathbf{p}_q$).** Serve as *anchor locations* in normalized coordinates $[0, 1]^2$ that ground the attention mechanism spatially. They represent the current prediction focus (e.g., the last predicted trajectory point) and provide spatial context for where to look in the feature maps.

**Sampling Offsets ($\boldsymbol{\Delta p}_{nkq}$).** Learned *displacement vectors* that adaptively shift attention away from the reference point toward informative spatial locations. These enable the model to dynamically focus on relevant scene elements (e.g., lane boundaries, nearby agents) rather than being constrained to a fixed grid.

**Attention Weights ($A_{nkq}$).** Determine the *importance* of each sampled location, allowing the model to emphasize the most relevant spatial features while suppressing irrelevant background information. The weights sum to 1 across all sampling points, creating a normalized spatial attention distribution.

The key innovation is that both offsets and weights are *content-dependent*—computed dynamically based on the current query state rather than being fixed. This enables efficient sparse attention that adapts to the spatial structure of each specific scene, focusing computational resources on the most informative regions.

## A.3 Gabor Filter Steerability

Gabor filters are defined by a sinusoidal plane wave modulated by a Gaussian envelope:

$$G_{\lambda,\theta,\psi,\sigma,\gamma}(x, y) = \exp\left(-\frac{x'^2+\gamma^2 y'^2}{2\sigma^2}\right) \cos\left(2\pi\frac{x'}{\lambda} + \psi\right), \tag{22}$$

where

$$x' = x\cos\theta + y\sin\theta, \quad y' = -x\sin\theta + y\cos\theta, \tag{23}$$

and $\lambda$ (wavelength), $\theta$ (orientation), $\psi$ (phase), $\sigma$ (scale), and $\gamma$ (aspect ratio) control the filter's frequency and spatial extent [41].

Crucially, Gabor filters are *steerable*, meaning any rotated version can be expressed as a linear combination of a finite set of basis filters:

$$G_{\lambda,\theta}(x,y) = \sum_{n=1}^{N} k_n(\theta)\, G_{\lambda,\theta_n}(x,y), \tag{24}$$

with fixed prototype orientations $\{\theta_n\}$ and interpolation coefficients $\{k_n(\theta)\}$ [26]. This property enables the network to capture directional patterns without having to learn separate filters for each orientation—leading to improved sample efficiency and built-in SO(2)-equivariance. In CASPNet, this steerability is leveraged to induce rotational quasi-equivariance in the first layers of the map encoder, ensuring that road elements are represented robustly regardless of global scene orientation. Additionally, by using filter banks that span multiple scales ($\sigma$) and aspect ratios ($\gamma$), the architecture gains partial SIM(2)-equivariance, responding consistently to rescaled and rotated map structures [26].

## A.4 The UniTraj Dataprocessing Pipeline

The pseudo-code in this section are mostly generated using [39].

**Phase 1: Temporal Window Extraction** The first processing stage extracts *uniformly sampled* windows containing historical context ($T_p$ steps) and future ground truth ($T_f$ steps) from raw agent trajectories. Frequency masking ensures consistent uniform temporal resolution with a sampling interval $\Delta t_s$ across datasets, and is combined with the original validity masks to indicate which observations are valid at each timestep. Validity masks indicate missing observations throughout the pipeline.

**Algorithm 2** Phase 1: Temporal Window Extraction

**Input:** Raw scenario tracks $\mathcal{T}$, time horizons $T_p, T_f$, sampling interval $\Delta t_s$
**Output:** Temporally windowed agent trajectories with validity masks

$T_{total} \leftarrow T_p + T_f$ *Total time window length*
$M_{freq} \leftarrow \text{generate\_mask}(T_p - 1, T_{total}, \Delta t_s)$ *Temporal sampling mask*
**for** each agent track $i \in \mathcal{T}$, timestep $t = 0$ to $T_{total}$ **do**
  Extract state vectors: $\boldsymbol{s}_i^{(t)} = [\boldsymbol{p}_i^{(t)}, l_i, w_i, h_i, \theta_i^{(t)}, \boldsymbol{v}_i^{(t)}, \text{valid}_i^{(t)}]^T$
  Apply temporal windowing: $\boldsymbol{s}_i \leftarrow \boldsymbol{s}_i[t_{start} : t_{start} + T_{total}]$
  Apply frequency masking: $\text{valid}_i^{(t)} \leftarrow \text{valid}_i^{(t)} \cdot M_{freq}[t]$
**end for**

**Phase 2: Map Feature Processing**

This phase converts heterogeneous map primitives (lanes, boundaries, signs, crosswalks) into standardized polyline sequences with consistent geometric

and semantic encoding. Polyline interpolation ensures uniform point density, direction vectors encode the local orientation of each segment, and type-based filtering selects relevant map elements for prediction scenarios.

**Algorithm 3** Phase 2: Map Feature Processing

**Input:** Raw map data $\mathcal{M}$, interpolation distance $d_{interp}$
**Output:** Standardized map polylines with geometric and semantic features

**for** each map element $m \in \mathcal{M}$ **do**
  Interpolate polyline points with uniform spacing $d_{interp}$
  Compute direction vectors: $\boldsymbol{d}_{k,l} = \boldsymbol{p}_{k,l+1} - \boldsymbol{p}_{k,l}$
  Assign semantic type encoding: $\boldsymbol{o}_{type} \in \{0,1\}^{20}$
  Store polyline: $\boldsymbol{L}_k = [\boldsymbol{p}_{k,l}, \boldsymbol{d}_{k,l}, \boldsymbol{o}_{type}]_{l=1}^{L}$
**end for**

### Phase 3: Agent Selection and Filtering

Agent filtering ensures that only relevant trajectories with sufficient motion and observation quality are retained for training. This phase applies distance-based motion thresholds, present-time validity requirements, and future continuity constraints to identify suitable center agents.

**Algorithm 4** Phase 3: Agent Selection and Filtering

**Input:** Agent trajectories $\mathcal{A}$, motion threshold $d_{min} = 2.0$m
**Output:** Filtered set of center agents $\mathcal{A}_{center}$

$\mathcal{A}_{center} \leftarrow \emptyset$
**for** each agent $i \in \mathcal{A}$ **do**
  Compute total motion: $\Delta d_i = \sum_{t=1}^{T_p-1} \|\boldsymbol{p}_i^{(t)} - \boldsymbol{p}_i^{(t-1)}\|_2$
  **if** $\Delta d_i \geq d_{min}$ AND $\text{valid}_i^{(T_p-1)} = 1$ **then**
    Add to center agents: $\mathcal{A}_{center} \leftarrow \mathcal{A}_{center} \cup \{i\}$
  **end if**
**end for**

### Phase 4: Coordinate System Transformation

For each center agent, the entire scene (including all other agents and map elements) is transformed to an agent-centric coordinate frame. This transformation consists of translation to center the agent's position at $t = T_p - 1$ at the origin, followed by rotation to align the agent's heading with the positive $x$-axis.

**Algorithm 5** Phase 4: Coordinate System Transformation

**Input:** Center agent $c$, all trajectories $\mathcal{A}$, map polylines $\mathcal{L}$
**Output:** Agent-centric coordinates for all elements

$\boldsymbol{p}_c^{ref} \leftarrow \boldsymbol{p}_c^{(T_p-1)}$ *Reference position*
$\theta_c^{ref} \leftarrow \theta_c^{(T_p-1)}$ *Reference heading*
**for** each agent $i \in \mathcal{A}$, timestep $t$ **do**
  Translate: $\boldsymbol{p}_i^{(t)} \leftarrow \boldsymbol{p}_i^{(t)} - \boldsymbol{p}_c^{ref}$
  Rotate: $\boldsymbol{p}_i^{(t)} \leftarrow \mathbf{R}(-\theta_c^{ref})\boldsymbol{p}_i^{(t)}$
  Transform heading: $\theta_i^{(t)} \leftarrow \theta_i^{(t)} - \theta_c^{ref}$
  Rotate velocity: $\boldsymbol{v}_i^{(t)} \leftarrow \mathbf{R}(-\theta_c^{ref})\boldsymbol{v}_i^{(t)}$
**end for**
**for** each polyline $k \in \mathcal{L}$, point $l$ **do**
  Apply same coordinate transformation to polyline points
  Filter polylines within spatial range: $\|\boldsymbol{p}_{polyline}\|_2 \leq \text{map\_range}$
**end for**

After this transformation, all spatial map features and agent states are expressed in a common reference frame, whose origin is the center agent's position at the last historical timestep $T_p - 1$. This yields both *translation* and *rotation* invariance, when perceiving the scene from the center agent's perspective.

**Phase 5: Feature Vector Assembly**

This phase constructs the full feature vectors for each agent by concatenating spatial state, agent type encoding, temporal step embedding, heading representation, and kinematic features. Invalid trajectory points are zeroed out according to the validity masks.

**Algorithm 6** Phase 5: Feature Vector Assembly

**Input:** Transformed agent states, validity masks
**Output:** Complete agent feature vectors $\boldsymbol{X}_d \in \mathbb{R}^{N_{\max} \times T_p \times F_{ap}}$

**for** each agent $i$, timestep $t$ **do**
  Spatial features: $\boldsymbol{f}_{spatial} = [\boldsymbol{p}_i^{(t)}, l_i, w_i, h_i] \in \mathbb{R}^6$
  Type encoding: $\boldsymbol{o}_{type} \in \{0,1\}^5$
  Time embedding: $\boldsymbol{e}_{time} \in \{0,1\}^{T_p+1}$
  Heading encoding: $\boldsymbol{h}_{embed} = [\sin(\theta_i^{(t)}), \cos(\theta_i^{(t)})] \in \mathbb{R}^2$
  Kinematic features: $\boldsymbol{f}_{kinematic} = [\boldsymbol{v}_i^{(t)}, \boldsymbol{a}_i^{(t)}] \in \mathbb{R}^4$
  Concatenate: $\boldsymbol{X}_d^{(i,t)} = [\boldsymbol{f}_{spatial}, \boldsymbol{o}_{type}, \boldsymbol{e}_{time}, \boldsymbol{h}_{embed}, \boldsymbol{f}_{kinematic}]$
  **if** $\text{valid}_i^{(t)} = 0$ **then**
    $\boldsymbol{X}_d^{(i,t)} \leftarrow \boldsymbol{0}$
  **end if**
**end for**

### Phase 6: Agent Proximity Filtering and Padding

To ensure computational tractability, the pipeline retains only the $N_{\max}$ closest agents to each center agent, where proximity is measured by the Euclidean distance at the final historical timestep. Agent features are then zero-padded to the fixed dimension $N_{\max}$ to enable efficient batch processing.

**Algorithm 7** Phase 6: Agent Proximity Filtering and Padding

**Input:** Agent features $\boldsymbol{X}_d$, maximum agents $N_{\max} = 64$
**Output:** Padded agent tensor $\boldsymbol{X}_d \in \mathbb{R}^{N_{\max} \times T_p \times F_{ap}}$

**for** each center agent $c$ **do**
  Compute distances: $d_{ic} = \|\boldsymbol{p}_i^{(T_p-1)} - \boldsymbol{p}_c^{(T_p-1)}\|_2$ for all agents $i$
  Select top-$N_{\max}$ closest agents by distance
  Zero-pad features to dimension $N_{\max} \times T_p \times F_{ap}$
  Create validity mask: $\boldsymbol{M}_d \in \{0,1\}^{N_{\max} \times T_p}$
**end for**

### Phase 7: Map Feature Processing

The agent-centric map polylines are converted into a fixed-size tensor representations in a three-stage process: segmentation based on geometric discontinuities (gaps $> 1.0$m), uniform resampling to exactly $L$ points per segment, and proximity-based selection of the top $K_{\max}$ segments closest to the center agent. Each point is encoded with geometric context including position, direction vectors, previous point reference, and semantic type information. The resulting map tensor $\boldsymbol{X}_s$ is padded to a fixed size of $K_{\max}$ segments, each with $L$ points, and a validity mask $\boldsymbol{M}_s$ is created to indicate

which segments are actually present.

**Algorithm 8** Phase 7: Map Feature Processing

**Input:** Agent-centric polylines, max polylines $K_{\max} = 256$, points per polyline $L = 20$

**Output:** Map tensor $\boldsymbol{X}_s \in \mathbb{R}^{K_{\max} \times L \times F_{map}}$, validity mask $\boldsymbol{M}_s$

1: **for** each polyline $k$ **do**
2: Segment polyline at geometric discontinuities (gaps $> 1.0$m)
3: Resample each segment to exactly $L$ uniform points
4: Compute geometric features: position, direction, previous point
5: Append semantic type encoding: $\boldsymbol{o}_{type} \in \{0, 1\}^{20}$
6: Assemble: $\boldsymbol{X}_s^{(k,l)} = [\text{position}, \text{direction}, \text{previous}, \boldsymbol{o}_{type}]$
7: **end for**
8: Select top-$K_{\max}$ polylines by proximity to center agent
9: Zero-pad to fixed dimensions and create validity mask

### Phase 8: Future Trajectory Processing

This phase processes the future trajectory ground truth for each agent and creates the center agent's target trajectory. Future trajectories are extracted, transformed to agent-centric coordinates, and validity masks are created to handle variable-length future observations.

**Algorithm 9** Phase 8: Future Trajectory Processing

**Input:** Future trajectory data, center agent indices

**Output:** Center ground truth $\boldsymbol{y}_c \in \mathbb{R}^{T_f \times 4}$, validity masks

1: **for** each agent $i$, future timestep $t \in [T_p, T_p + T_f)$ **do**
2: Extract future state: $\boldsymbol{s}_i^{(t)} = [\boldsymbol{p}_i^{(t)}, \boldsymbol{v}_i^{(t)}]$
3: Apply agent-centric transformation
4: Store in future tensor: $\tilde{\boldsymbol{X}}_d[i, t - T_p] = \boldsymbol{s}_i^{(t)}$
5: **end for**
6: **for** each center agent $c$ **do**
7: Extract center ground truth: $\boldsymbol{y}_c = \tilde{\boldsymbol{X}}_d[c, :, :]$
8: Create future validity mask: $\tilde{\boldsymbol{M}}_d \in \{0, 1\}^{T_f}$
9: Compute final valid index: $idx_{final} = \max\{t : \tilde{\boldsymbol{M}}_d[t] = 1\}$
10: **end for**

### Phase 9: DatasetItem Assembly

The final processing phase assembles all processed components into final `DatasetItem` instances. It performs data validation, applies optional attribute masking (e.g., zeroing z-coordinates or object bounding boxes),

ensures float32 compatibility, and finally creates structured `DatasetItem` instances.

**Algorithm 10** Phase 9: DatasetItem Assembly

**Input:** All processed tensors and masks
**Output:** Final `DatasetItem` instance
1: Create `DatasetItem` instance with all processed tensors
2: Apply attribute masking (optional): zero out z-axis, size, velocity, etc.
3: Convert floating data types to float32
4: **return** `DatasetItem` instance

The resulting `DatasetItem`'s are subsequently saved to disk for training and evaluation. They include all agent tensors, map tensors, ground truth labels, and validity masks as `numpy` arrays in various formats for easy handling at different training, evaluation or visualization stages. The `DatasetItem` class serves as the primary data container for processed trajectory scenarios. Figure 19 shows the complete structure of this class, including all tensor attributes, metadata fields, and utility methods used throughout the training and evaluation pipeline.


Figure 19: Class diagram of the `DatasetItem` structure showing all tensor attributes, metadata fields, and utility methods. This class encapsulates the processed scenario data as described in subsection A.4.

The `DatasetItem` contains the standardized tensor representations described in Table 4, including agent trajectories (`obj_trajs`), map polylines (`map_polylines`), ground truth targets (`center_gt_trajs`), and their corresponding validity masks. The resulting `DatasetItem`'s are subsequently saved to disk for training and evaluation. They include all agent tensors, map tensors, ground truth labels, and validity masks as `numpy` arrays in various formats for easy handling at different training, evaluation or visualization stages.

Figure 20 illustrates the PyTorch Lightning data module architecture used for batch processing and data loading. The `LitDatamodule` and its configuration class manage the entire data pipeline from raw scenarios to training-ready batches.

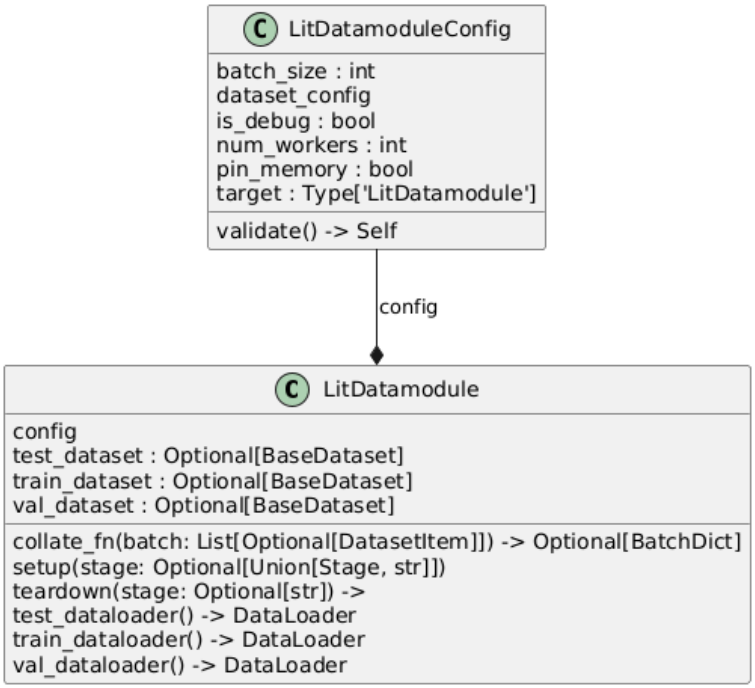



Figure 20: Class diagram of the data processing components showing the `LitDatamodule` and `LitDatamoduleConfig` classes. These components orchestrate the data loading pipeline described in subsection A.4.

The `LitDatamodule` implements the PyTorch Lightning data module interface, providing standardized train, validation, and test data loaders with configurable batch sizes, worker processes, and memory pinning options as specified in the `LitDatamoduleConfig`.

## A.5 UniTraj Directory Structure

```
unitraj/
├── __init__.py
├── configs/
│   ├── experiment_config.py
│   │   └── ExperimentConfig | top-level experiment config
│   ├── path_config.py
│   │   └── PathConfig | centralized Singleton for path-handling
│   └── wandb_config.py
│       └── WandBConfig | WandB integration for PyTorch Lightning
└── datasets/
    ├── base_dataparser.py
    │   └── BaseDataParser - multiprocessing, file & metadata handling
    └── dataparser.py
```

```
    ├── DataParser | Processing pipeline as per subsection A.4
    ├── base_dataset.py | BaseDataset wrapper
    │   └── BaseDataset | PyTorch dataset
    ├── common_utils.py | shared parsers & transforms
    ├── types.py | type definitions
    │   ├── DatasetItem | final dataset item structure
    │   └── BatchInputDict | collated tensor dict
├── lightning/
    ├── lit_datamodule.py | LightningDataModule
    │   └── LitDatamodule | PyTorch Lightning data module
    ├── lit_trainer_factory.py | pl.Trainer factory
    │   └── TrainerFactory | factory for pl.Trainer
├── models/
    ├── base_model.py | BaseModel, BaseModelConfig
    │   └── BaseModel | base pl.LightningModule for UniTraj
├── console.py
    ├── Console | Rich console for logging
├── base_config.py
    ├── BaseConfig | abstract base config for Config-as-Factory
    │   pattern
├── visualization.py | plotting and rendering helpers
    ├── plot_dataset_item() | static visualization for single DatasetItem
    ├── check_loaded_data() | data integrity visualization check
    ├── visualize_batch_data() | batch data visualization
    ├── concatenate_images() | image concatenation utility
    ├── concatenate_varying() | varying size image concatenation
    ├── visualize_prediction() | prediction result visualization
```

# References


[1] Yihan Hu et al. "Planning-oriented autonomous driving". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2023, pp. 17853–17862.

[2] Harsh Yadav et al. *LMFormer: Lane based Motion Prediction Transformer*. 2025. arXiv: 2504.10275 [cs.CV]. URL: https://arxiv.org/abs/2504.10275.

[3] Zikang Zhou et al. "QCNeXt: A Next-Generation Framework For Joint Multi-Agent Trajectory Prediction". In: *arXiv preprint arXiv:2306.10508* (2023).

[4] Holger Caesar et al. *nuScenes: A multimodal dataset for autonomous driving*. 2020. arXiv: 1903.11027 [cs.LG]. URL: https://arxiv.org/abs/1903.11027.

[5] Benjamin Wilson et al. *Argoverse 2: Next Generation Datasets for Self-Driving Perception and Forecasting*. 2023. arXiv: 2301.00493 [cs.CV]. URL: https://arxiv.org/abs/2301.00493.

[6] Pei Sun et al. *Scalability in Perception for Autonomous Driving: Waymo Open Dataset*. 2020. arXiv: 1912.04838 [cs.CV]. URL: https://arxiv.org/abs/1912.04838.

[7] Lan Feng et al. *UniTraj: A Unified Framework for Scalable Vehicle Trajectory Prediction*. 2024. arXiv: 2403.15098 [cs.CV]. URL: https://arxiv.org/abs/2403.15098.

[8] Zikang Zhou et al. "Query-Centric Trajectory Prediction". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2023.

[9] Maximilian Schäfer et al. *Context-Aware Scene Prediction Network (CASPNet)*. 2022. arXiv: 2201.06933 [cs.CV]. URL: https://arxiv.org/abs/2201.06933.

[10] Shaoshuai Shi et al. *Motion Transformer with Global Intention Localization and Local Movement Refinement*. 2022. arXiv: 2209.13508 [cs.CV]. URL: https://arxiv.org/abs/2209.13508.

[11] Jiyang Gao et al. "Vectornet: Encoding hd maps and agent dynamics from vectorized representation". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 11525–11533.

[12] Henggang Cui et al. "Multimodal trajectory predictions for autonomous driving using deep convolutional networks". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 2090–2096.

[13] Yuning Chai et al. "MultiPath: Multiple Probabilistic Anchor Trajectory Hypotheses for Behavior Prediction". In: *CoRR* abs/1910.05449 (2019). arXiv: 1910.05449. URL: http://arxiv.org/abs/1910.05449.

[14] Maximilian Schäfer, Kun Zhao, and Anton Kummert. *CASPNet++: Joint Multi-Agent Motion Prediction*. 2023. arXiv: 2308.07751 [cs.CV]. URL: https://arxiv.org/abs/2308.07751.

[15] Harsh Yadav et al. "CASPFormer: Trajectory Prediction from BEV Images with Deformable Attention". In: *Pattern Recognition*. Springer Nature Switzerland, Dec. 2024, pp. 420–434. ISBN: 9783031784477. DOI: 10.1007/978-3-031-78447-7_28. URL: http://dx.doi.org/10.1007/978-3-031-78447-7_28.

[16] Ming Liang et al. "Learning lane graph representations for motion forecasting". In: *European conference on computer vision*. Springer. 2020, pp. 541–556.

[17] Jiyang Gao et al. *VectorNet: Encoding HD Maps and Agent Dynamics from Vectorized Representation*. 2020. arXiv: 2005.04259 [cs.CV]. URL: https://arxiv.org/abs/2005.04259.

[18] Zikang Zhou et al. "Hivt: Hierarchical vector transformer for multi-agent motion prediction". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2022, pp. 8683–8693.

[19] Michael M. Bronstein et al. *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges*. 2021. arXiv: 2104.13478 [cs.LG]. URL: https://arxiv.org/abs/2104.13478.

[20] Yang Li et al. *Learnable Fourier Features for Multi-Dimensional Spatial Positional Encoding*. 2021. arXiv: 2106.02795 [cs.LG]. URL: https://arxiv.org/abs/2106.02795.

[21] Quanyi Li et al. *ScenarioNet: Open-Source Platform for Large-Scale Traffic Scenario Simulation and Modeling*. 2023. arXiv: 2306.12241 [cs.RO]. URL: https://arxiv.org/abs/2306.12241.

[22] Scott Ettinger et al. *Large Scale Interactive Motion Forecasting for Autonomous Driving: The Waymo Open Motion Dataset*. 2021. arXiv: 2104.10133 [cs.CV]. URL: https://arxiv.org/abs/2104.10133.

[23] Tsung-Yi Lin et al. *Feature Pyramid Networks for Object Detection*. 2017. arXiv: 1612.03144 [cs.CV]. URL: https://arxiv.org/abs/1612.03144.

[24] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV]. URL: https://arxiv.org/abs/1505.04597.

[25] Ozan Oktay et al. *Attention U-Net: Learning Where to Look for the Pancreas*. 2018. arXiv: 1804.03999 [cs.CV]. URL: https://arxiv.org/abs/1804.03999.

[26] I. Kalliomäki and J. Lampinen. "Approximate Steerability of Gabor Filters for Feature Detection". In: *Image Analysis*. Ed. by Heikki Kalviainen, Jussi Parkkinen, and Arto Kaarna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 940–949. ISBN: 978-3-540-31566-7.

[27] Xingjian Shi et al. *Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting*. 2015. arXiv: 1506.04214 [cs.CV]. URL: https://arxiv.org/abs/1506.04214.

[28] Ismail Khalfaoui-Hassani, Thomas Pellegrini, and Timothée Masquelier. *Dilated convolution with learnable spacings*. 2023. arXiv: 2112.03740 [cs.CV]. URL: https://arxiv.org/abs/2112.03740.

[29] Xizhou Zhu et al. *Deformable DETR: Deformable Transformers for End-to-End Object Detection*. 2021. arXiv: 2010.04159 [cs.CV]. URL: https://arxiv.org/abs/2010.04159.

[30] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[31] Christian Rupprecht et al. "Learning in an uncertain world: Representing ambiguity through multiple hypotheses". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 3591–3600.

[32] Charles R. Qi et al. *PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation*. 2017. arXiv: 1612.00593 [cs.CV]. URL: https://arxiv.org/abs/1612.00593.

[33] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[34] Nicolas Carion et al. *End-to-End Object Detection with Transformers*. 2020. arXiv: 2005.12872 [cs.CV]. URL: https://arxiv.org/abs/2005.12872.

[35] Jiquan Ngiam et al. *Scene Transformer: A unified architecture for predicting future trajectories of multiple agents*. 2022. arXiv: 2106.08417 [cs.CV]. URL: https://arxiv.org/abs/2106.08417.

[36] Christopher M. Bishop. *Mixture Density Networks*. Tech. rep. NCRG/94/004. Birmingham, UK: Aston University, 1994. URL: http://publications.aston.ac.uk/id/eprint/373/.

[37] Shaoshuai Shi et al. *MTR++: Multi-Agent Motion Prediction with Symmetric Scene Modeling and Guided Intention Querying*. 2023. arXiv: 2306.17770 [cs.CV]. URL: https://arxiv.org/abs/2306.17770.

[38] Shilong Liu et al. "DAB-DETR: Dynamic Anchor Boxes are Better Queries for DETR". In: *International Conference on Learning Representations*. 2022. URL: https://openreview.net/forum?id=oMI9PjOb9Jl.

[39] GitHub, Anthropic. *GitHub Copilot using Claude Sonnet 4 by Anthropic*. 2025. URL: https://github.com/features/copilot.

[40] Quanyi Li et al. *MetaDrive: Composing Diverse Driving Scenarios for Generalizable Reinforcement Learning*. 2022. arXiv: 2109.12674 [cs.LG]. URL: https://arxiv.org/abs/2109.12674.

[41] Shangzhen Luan et al. "Gabor Convolutional Networks". In: *IEEE Transactions on Image Processing* 27.9 (Sept. 2018), pp. 4357–4366. ISSN: 1941-0042. DOI: 10.1109/tip.2018.2835143. URL: http://dx.doi.org/10.1109/TIP.2018.2835143.