BitTP: The Lightweight Trajectory Prediction Model with BitLLM for Edge-Devices

Supplementary Material

S1. Implementation Details and Evaluation Metrics

This section provides a comprehensive breakdown of the model architecture integration, the specific metrics used for performance assessment, and the hardware protocols employed for efficiency benchmarking.

S1.1. Backbone and BitLinear Integration

We expand on the implementation details of the BitTP architecture described in the main paper. Our backbone utilizes the **T5-small** encoder-decoder structure, consisting of 6 encoder blocks and 6 decoder blocks ($d_{model} = 512, d_{ff} = 2048, \text{heads} = 8$).

Quantization Injection Points. We apply our recursive replacement algorithm to inject `BitLinear` modules into specific sub-layers of the Transformer blocks. The target modules include:

- **Attention Projections:** The Query ($\mathbf{W}_Q$), Key ($\mathbf{W}_K$), Value ($\mathbf{W}_V$), and Output ($\mathbf{W}_O$) matrices in both Self-Attention and Cross-Attention mechanisms.
- **Feed-Forward Networks (FFN):** The expansion ($\mathbf{W}_i$) and projection ($\mathbf{W}_o$) layers within the position-wise FFNs.

S1.2. Latency and Memory Measurement Protocol

To ensure fair and reproducible comparisons across different quantization strategies, we conducted all efficiency measurements under a standardized environment that reflects our experimental configuration.

- **Computing Device:** All benchmarks were performed on a single **NVIDIA RTX 3090** GPU with 24GB of VRAM.
- **Batch Size:** We set the inference batch size to $B = 1$
- **Decoding Settings:** Unless stated otherwise, measurements include the autoregressive generation time for the full prediction horizon $T_{\text{pred}} = 12$. Crucially, we employed **stochastic sampling** with a temperature parameter $\tau = 0.7$ (rather than greedy decoding) to account for the computational overhead of the random sampling process required for generating diverse trajectories.
- **Precision & Cache:** Computations were performed in mixed precision (BF16/FP32) with the Key-Value (KV) cache enabled to optimize sequential decoding throughput.

S1.3. Evaluation Metrics

We evaluate the predictive performance using standard trajectory prediction metrics. Since our model generates multimodal predictions via stochastic sampling, we report the Best-of-K metrics, where the sample closest to the ground truth is selected for evaluation. In all experiments, we use $K = 20$ stochastic samples per prediction. 1000 trajectory samples are generated for each input in the main experiment, while only 1 sample per input is used in the CPU-based experiments in Sec. S5.

Let $\mathbf{Y}_i \in \mathbb{R}^{T\times 2}$ be the ground truth trajectory for pedestrian i, and $\hat{\mathbf{Y}}_i^{(k)}$ be the k-th predicted sample among K generated trajectories. If the dataset provides homography matrices H, all coordinates are projected from pixel space to world coordinates (meters) via a mapping function $\mathcal{M}(\cdot, H)$ before error calculation.

- **Average Displacement Error (ADE):** The mean Euclidean distance between the predicted path and the ground truth over all time steps. We report the minimum ADE (minADE) across K samples, averaged over all N pedestrians in the dataset:

$$\text{ADE} = \frac{1}{N}\sum_{i=1}^{N}\min_{k=1}^{K}\left(\frac{1}{T}\sum_{t=1}^{T}\|\hat{\mathbf{y}}_{t,i}^{(k)} - \mathbf{y}_{t,i}\|_2\right) \tag{S1}$$

- **Final Displacement Error (FDE):** The Euclidean distance between the predicted position and the ground truth at the final time step T. Similarly, we report the minimum FDE (minFDE):

$$\text{FDE} = \frac{1}{N}\sum_{i=1}^{N}\min_{k=1}^{K}\left(\|\hat{\mathbf{y}}_{T,i}^{(k)} - \mathbf{y}_{T,i}\|_2\right) \tag{S2}$$

- **Peak Memory:** The peak VRAM usage required to load the model parameters, measured in Megabytes (MB). This metric directly reflects the compression ratio achieved by quantization.
- **Inference Latency:** The average wall-clock time (milliseconds) required to generate a full trajectory ($T = 12$) for a single agent ($B = 1$).

S1.4. Latency and Memory Measurement Protocol

To ensure fair and reproducible comparisons across different quantization strategies, we conducted all efficiency measurements under a standardized environment that reflects our experimental configuration.

- **Computing Device:** All benchmarks were performed on a single **NVIDIA RTX 3090 (24GB)** GPU.
- **Batch Size:** We utilized a batch size of 1, online trajectory prediction scenario

Table S1. **Wall-Clock Training Time Comparison across All Datasets.** Measured on a single NVIDIA RTX 3090 with a fixed batch size of 128 and learning rate of 1×10^{-4} for 8 epochs. The symbol $\times$ denotes the relative time increase compared to the baseline.

Model Variant	Target	Training Duration (Hours)					Avg. Overhead
		ETH	Hotel	Univ	Zara1	Zara2	
LMTraj-SUP (Baseline)	BF16	40h	33h	9h	31h	28h	1.00×
BitTP-Both	Both	72h	60h	18h	56h	52h	1.90×
BitTP-Activ	Activations only	68h	59h	18h	55h	52h	1.85×
BitTP-Weight	Weights only	46h	40h	12h	37h	35h	1.20×

- **Decoding Settings:** Unless stated otherwise, measurements include the autoregressive generation time for the full prediction horizon ($T_{pred} = 12$). Crucially, we employed **stochastic sampling** with a temperature of $\tau = 0.7$ (rather than greedy decoding) to account for the computational overhead of the random sampling process required for generating diverse trajectories.
- **Precision & Cache:** Computations were performed in mixed precision (BF16/FP32) with the Key-Value (KV) cache enabled to optimize sequential decoding throughput.

S2. Quantization Methods: Existing Techniques

This section elaborates on the theoretical underpinnings and implementation details of the quantization strategies compared in this study. For the post-training quantization (PTQ) baselines, we leverage the optimized kernels provided by the `bitsandbytes` library [10, 12], which implement vector-wise quantization and normal-float data types. We contrast these with our proposed BitTP, which fundamentally alters the linear projection mechanics.

S2.1. INT8 Baseline: Vector-wise Quantization with Decomposition

For the 8-bit baseline, we employ the **LLM.int8()** methodology. Unlike standard abs-max quantization which can be degraded by outlier features, this method utilizes a *mixed-precision decomposition* strategy.

Vector-wise Quantization. The matrix multiplication $\mathbf{Y} = \mathbf{X}\mathbf{W}$ is approximated by scaling rows of $\mathbf{X}$ and columns of $\mathbf{W}$. Let C_x and C_w be the scaling constants for the input and weight tensors, respectively. The quantized computation is defined as:

$$\mathbf{Y} \approx \text{dequant}\left(\text{quant}(\mathbf{X}) \cdot \text{quant}(\mathbf{W})\right) = S_x \cdot \mathbf{X}_{\text{int8}} \cdot \mathbf{W}_{\text{int8}} \cdot S_w^T \tag{S3}$$

where S_x and S_w preserve the dynamic range of the activations and weights.

Outlier Handling (Decomposition). To handle emergence of extreme outliers in Transformer activations, the matrix multiplication is decomposed into two parts based on an outlier threshold α:

$$\mathbf{Y} = \underbrace{(\mathbf{X}_{out} \cdot \mathbf{W}_{out})}_{\text{BF16}} + \underbrace{(\mathbf{X}_{int} \cdot \mathbf{W}_{int})}_{\text{INT8}} \tag{S4}$$

Dimensions where $|x_i| > \alpha$ are extracted and computed in high precision (BF16), while the bulk of dimensions are computed in INT8. This ensures numerical stability but introduces the inference latency overhead observed in our experiments.

S2.2. 4-bit Baseline: NormalFloat (NF4) Quantization

For the 4-bit baseline, we adopt the **NormalFloat4 (NF4)** data type. This approach assumes that the weights of a converged neural network follow a zero-mean normal distribution $\mathcal{N}(0, 1)$ and employs *Quantile Quantization*.

Information-Theoretic Optimality. Standard integer quantization relies on linear spacing, which is suboptimal for bell-curved distributions. NF4 constructs quantization levels q_i such that each bin has equal probability mass. The levels are determined by the quantiles of the normal distribution:

$$q_i = Q^{-1}\left(\frac{i}{2^k + 1}\right) \tag{S5}$$

where Q^{-1} is the inverse cumulative distribution function (CDF) of $\mathcal{N}(0, 1)$ and $k = 4$ bits. During the forward pass, these 4-bit stored weights are dequantized on-the-fly to BF16 for computation, prioritizing memory efficiency over arithmetic speedup.

S3. Training-Time Behavior

In this section, we analyze the optimization dynamics of training BitTP variants from scratch (or fine-tuning) compared to the BF16 baseline. We focus on training stability and computational overhead. While the primary goal of BitTP is to reduce inference memory and latency, analyzing training efficiency is crucial for understanding the cost of using `BitLinear`.

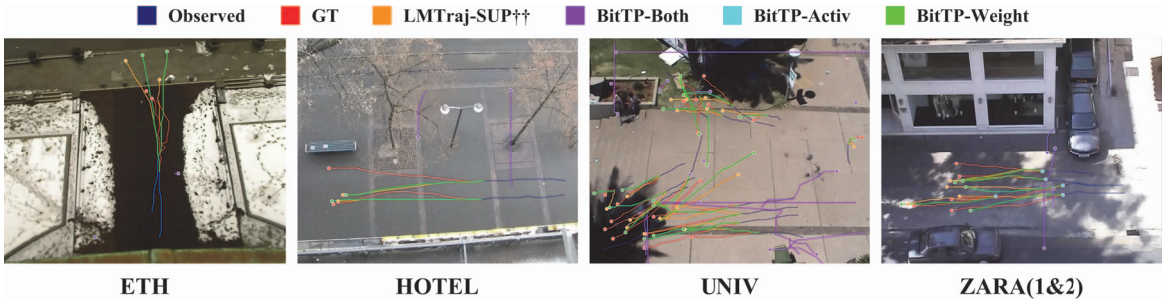

Figure S1. **Qualitative comparison of predicted pedestrian trajectories on the ETH/UCY scenes (ETH, HOTEL, UNIV, ZARA1&2) under different quantization strategies.** Observed past trajectories, ground-truth futures, and predictions from LMTraj-SUP††, BitTP-Both, BitTP-Activ, and BitTP-Weight are overlaid; BitTP-Activ frequently collapses and fails to produce plausible future trajectories, while BitTP-Weight preserves accurate trajectory shapes.

Measurement Setup. We measured the wall-clock training time on a single **NVIDIA RTX 3090 (24GB)** GPU. The training configuration was standardized with a batch size of 128 and a learning rate of 1×10^{-4} over 8 epochs.

Training Duration and Overhead. Tab. S1 presents the total training duration required across five datasets (ETH, Hotel, Univ, Zara1, Zara2). We observe distinct overhead patterns depending on the quantization target:

- **Weight-Only Efficiency:** The *BitTP-Weight* variant incurs a relatively minor overhead (avg. $\sim$ 20%) compared to the baseline. This is because weight quantization (ternarization) involves element-wise operations that are computationally inexpensive relative to the full matrix multiplication cost.
- **Activation Quantization Cost:** The *BitTP-Both* and *BitTP-Activ* variants show a significant increase in training time ($\sim$ 1.8–1.9×). This substantial overhead is attributed to the **dynamic activation quantization** process. Unlike weights, activations change every forward pass, requiring on-the-fly calculation of scaling factors (e.g., AbsMean) and rounding operations for every token. In our Python-based implementation, this creates a computational bottleneck that limits training throughput.

Consistency Across Datasets. This trend was consistent across all datasets. For instance, in the Univ dataset, training BitTP-Weight required 12 hours, whereas BitTP-Both required 18 hours (1.5×), reinforcing the observation that the computational bottleneck lies primarily in the activation quantization path.

S4. Qualitative Results of Forecasting the Trajectory

This section provides qualitative analyses of language-conditioned trajectory forecasting under different quantization strategies. Fig. S1-S3 visualize stochastic predictions on the ETH/UCY benchmark and compare the full-precision LMTraj-SUP†† model with BitTP-Weight, BitTP-Both, and BitTP-Activ. Together, these figures illustrate how weight-only quantization preserves scene-consistent motion, whereas activation quantization leads to severe degradation in both trajectory generation and language outputs, complementing the quantitative results in the main paper.

Multi-scene comparison. Fig. S1 summarizes qualitative trajectory forecasts across five scenes (ETH, HOTEL, UNIV, ZARA1&2) under different quantization strategies. In all scenes, LMTraj-SUP†† and BitTP-Weight produce future trajectories that closely align with the ground-truth (GT). In contrast, BitTP-Both and BitTP-Activ fail to match the GT in a much more drastic manner. BitTP-Both still generates trajectories; however, the predicted paths are often clearly unrealistic and significantly displaced from the GT routes, sometimes bending into unrealistic directions or overshooting the intended goal. BitTP-Activ exhibits the most severe failure modes: for many examples, it does not produce a usable trajectory at all, either terminating without valid output or entering an effectively unbounded generation regime. The ZARA example is particularly illustrative—although points are plotted, they stem from such runaway decoding that no coherent trajectory can be drawn.

Input and output in the forecasting task. Fig. S2 and S3 decompose the qualitative results from Fig. S1 by explicitly showing the language-based input and output for representative examples on the ETH/HOTEL and UNIV/ZARA scenes. For each case, the observation is formatted as a question–context pair: the context encodes the past trajectory as a token sequence, and the question requests the future motion of pedestrian 0 over the forecasting horizon. The figures then display both the generated text answer and the corresponding trajectory tokens rendered in the scene. LMTraj-SUP†† and BitTP-Weight produce well-structured answers that correctly refer to the requested forecast (e.g.,

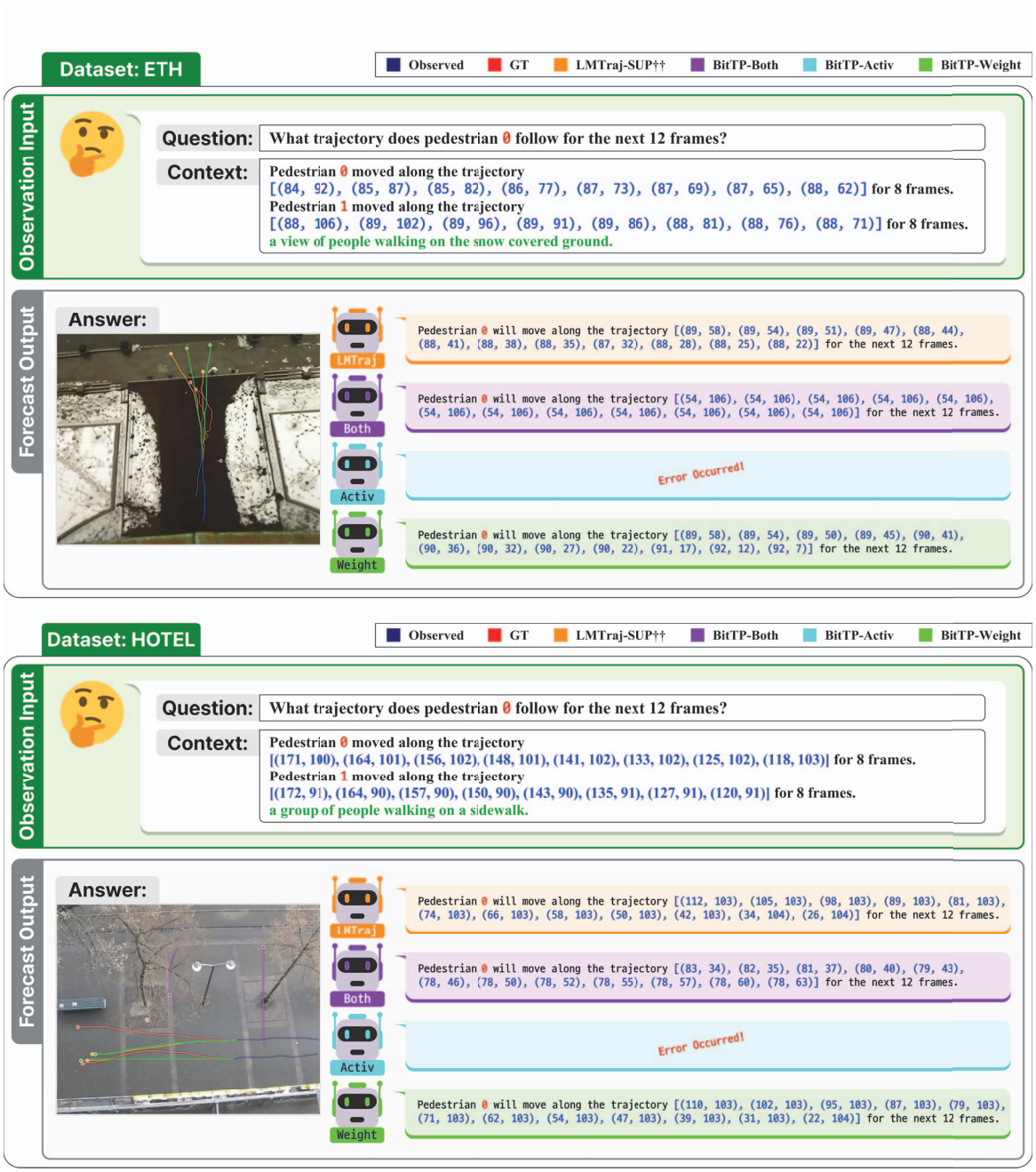

Figure S2. **Qualitative comparison of predicted trajectories on the ETH and HOTEL scenes under different quantization strategies for the trajectory forecast task.** For each example, the input observation consists of a question and a context. Only outputs for pedestrian 0 are visualized. Red denotes the pedestrian number, blue denotes the predicted trajectory, and green denotes the context; all colored elements are placeholders that can be replaced.

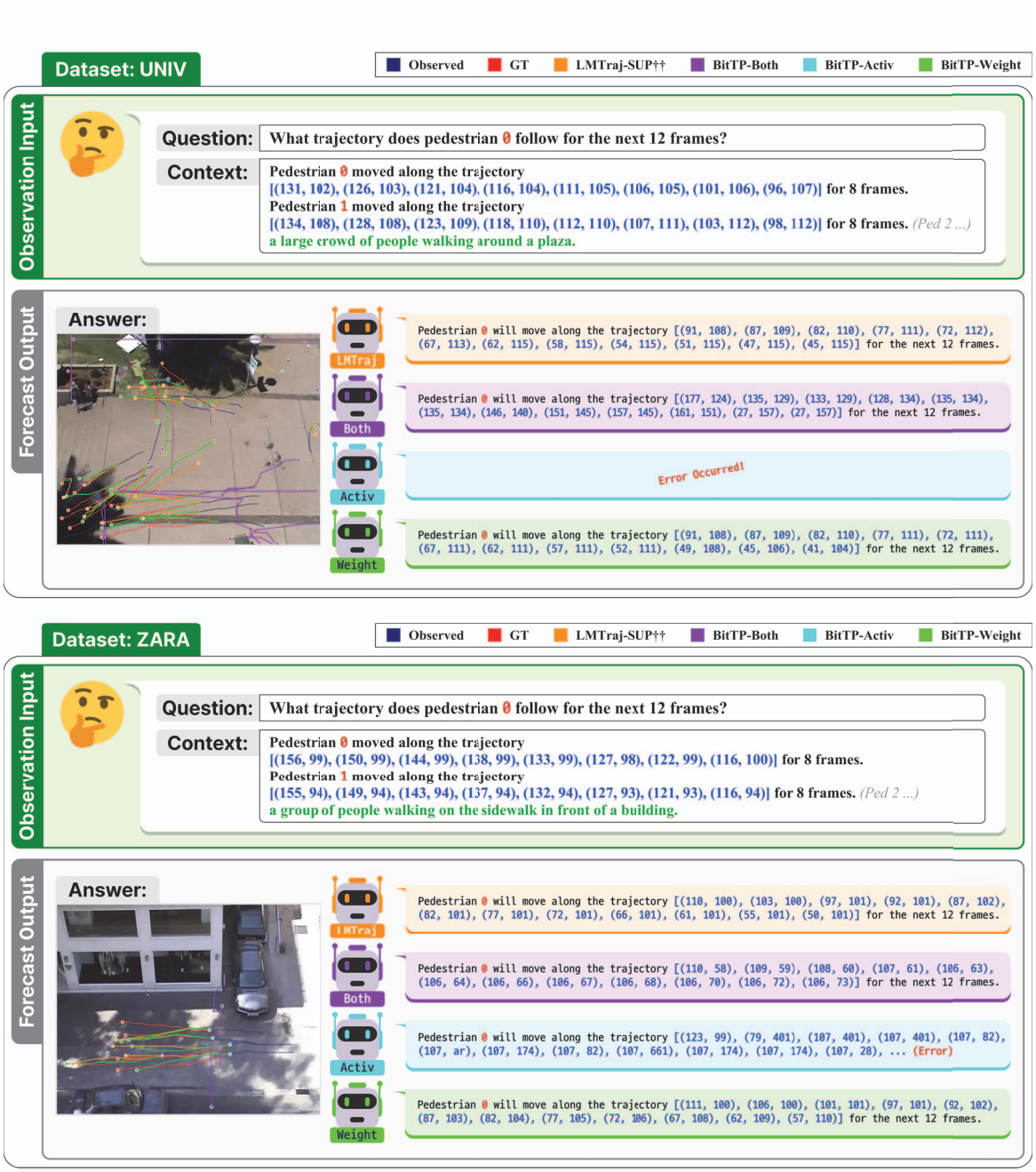

Figure S3. **Qualitative comparison of predicted trajectories on the UNIV and ZARA scenes under different quantization strategies for the trajectory forecast task.** For each example, the input observation consists of a question and a context. Only outputs for pedestrian 0 are visualized. Red denotes the pedestrian number, blue denotes the predicted trajectory, and green denotes the context; all colored elements are placeholders that can be replaced.

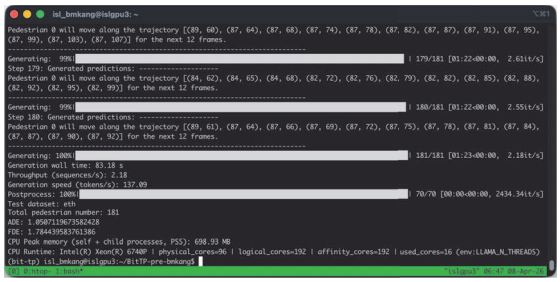

Figure S4. **Real-time execution log of BitTP-Weight in a CPU-only environment.** The screenshot displays the terminal output during inference on the ETH dataset using the `llama.cpp` framework, including runtime metrics and hardware utilization (Intel Xeon 6740P). Qualitative video demonstrations are available at: https://mintcat98.github.io/BitTP/

horizon length, direction of movement) and generate coordinate tokens that form a coherent, scene-consistent trajectory when visualized. In contrast, BitTP-Both and BitTP-Activ reveal how activation-involving quantization corrupts the decoding process itself: the outputs contain incomplete or repetitive phrases, or excessively long token streams that never settle into a valid trajectory sequence. Fig. S2 and S3, therefore, complement Fig. S1 by making explicit how the same failure patterns seen at the trajectory level already manifest at the language-interface level, while BitTP-Weight maintains reliable question–answer behavior and accurate future-path generation.

S5. CPU-based Inference and Cost Evaluation

Fig. S4 presents the execution log of BitTP in a CPU-only environment, exploring the feasibility of deploying an LLM-based predictor under strict hardware constraints. Despite the inherently massive computational requirements of LLMs, by leveraging Quantization-Aware Training (QAT) and a specialized **TQ1 kernel** [46] within the `llama.cpp` [18] framework, BitTP-Weight successfully achieves a throughput of 2.18 sequences/s on the ETH subset. This result confirms that running sophisticated trajectory prediction models entirely on resource-constrained on-board computers without GPU acceleration is practically possible. To measure realistic edge-device latency, our CPU evaluation uses single-sample inference, departing from the standard 1,000-sample GPU protocol used for reproducible benchmarking. While this naturally introduces slight stochastic variance, it effectively demonstrates the model's practical execution speed under strict resource constraints.

Furthermore, the execution log directly validates our memory efficiency analysis: the peak CPU memory consumption (PSS) was measured at only 698.93 MB (including child processes), confirming the exceptionally lightweight memory footprint of our ternary-quantized model. Collectively, these results demonstrate that BitTP-Weight, when aggressively quantized and paired with a highly-optimized inference engine, significantly lowers the hardware barrier, opening up new possibilities for deploying LLMs on CPU-bound edge platforms.