FROM CONTRACTION TO CHAOS:
COMPOUNDING ERROR RESISTANT WORLD MODELS FOR ROBOTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Learned world models are usually rolled out autoregressively: each prediction is fed back as input, so small one-step errors compound over the horizon. How fast they compound is determined by the system's sensitivity to initial conditions, one of the three mathematical properties of chaos. Under a tracking controller, deviations are corrected, and errors stay bounded; in a chaotic system, they grow exponentially. Auto-regressive (AR) models, such as the temporal convolutional networks widely used in mobile robotics, learn a simple one-step map but pay the full cost of compounding. One-shot Multi-step (MS) models never feed back their own errors, which theory predicts should pay off over long horizons. However, they cannot reuse what they learn for one step when predicting the next, so they need more data to learn the multi-step map. Neither class dominates across the sensitivity spectrum. We propose Multi-Step Temporal Mixture Probabilistic (MTM-Pro), a hybrid world model trained offline on logged trajectories with two coupled objectives: an MS objective that fuses overlapping multi-step forecasts through a learned temporal mixture, and an AR objective that rolls the model out on its own predictions. We evaluate on five datasets chosen to span the sensitivity spectrum: closed-loop quadcopter tracking on nominal trajectories, on aggressive trajectories, and at the edge of the control envelope; open-loop driving of an off-road uncrewed ground vehicle (UGV); and the chaotic Lorenz system. AR baselines degrade as sensitivity increases, while MS baselines become competitive only on the most sensitive system. MTM-Pro matches the best baseline on the least sensitive datasets and reduces 5-second drift by more than three orders of magnitude on Lorenz, making it the only model reliable across the whole spectrum. Our code and links to all datasets are publicly available at https://anonymous.4open.science/r/MTM-Pro.

1 INTRODUCTION

In learned robot dynamics modeling (i.e., world models), *compounded prediction error* remains an open problem. Once a robot is deployed, every prediction error surfaces inside its world model and compounds over time. How long the robot can cope with this growing error is what separates a catastrophic failure from resilient autonomy. Concretely, compounded error bounds three practical quantities: **(a)** how long a deployed controller can keep operating without fresh or accurate observations (e.g., heterogeneous sensor rates, sensor failure, system lag), **(b)** how far the online imagined rollouts of a Model Predictive Control (MPC) can reliably look ahead, and **(c)** how far training rollouts can be unrolled while remaining realistic in Model-based Deep Reinforcement Learning (MBDRL). Yet, single-step or short-horizon evaluation gives a false sense of reliability (Lambert et al., 2021): when previous predictions are fed back as new observations over a longer horizon, small errors accumulate and the very same model exhibits a radically different behavior. The critical consequence for learning-based models is *self-induced drift*, where compounded errors gradually push the trajectory wildly off course.[1] Therefore, we ask *how reliable is a model over multiple time-steps when it does not have access to ground truth observations or states?*

How fast errors compound is not a property of the model alone but of the dynamics being modeled. Even a model that is exact beyond the first step will diverge from the true trajectory if that first

[1] We are not addressing *robust control* (i.e., robustness to bounded error), but model resilience over multi-step horizons.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

step is slightly wrong; by how much is set by the system's sensitivity to initial conditions. Robotic data span a whole spectrum of such sensitivity, as exemplified on the left side of Fig. 1. When trajectories are recorded under a tracking controller, deviations are corrected in the logged actions, so the future remains largely readable from the recent history and the dynamics are *contracting*. When commands are applied open-loop, nothing corrects a deviation, which persists and *accumulates*. In a chaotic system such as the Lorenz attractor (Lorenz, 1963), any deviation grows *exponentially* into a completely different future. Therefore, compounding error grows from bounded, to accumulating, to exponential along this spectrum. A world model deployed across regimes must face this whole spectrum with a single architecture.

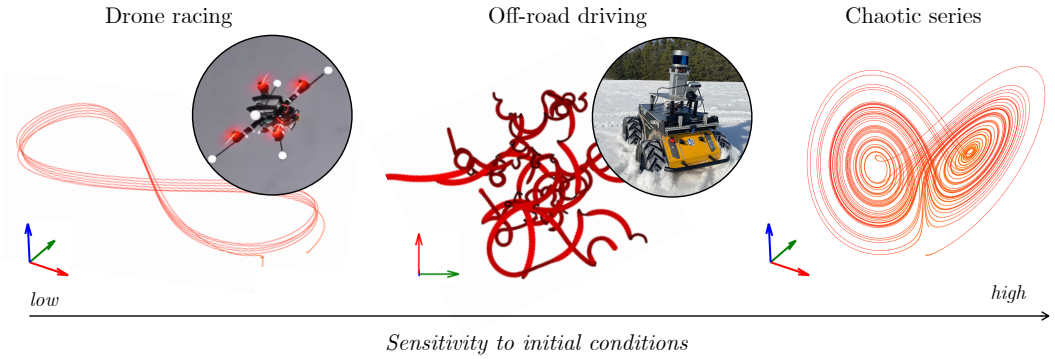


Figure 1: A subset of three trajectories on which we want to predict the future states over a short horizon: a drone repeating the same pattern simplifies the prediction; a rover randomly changing direction gets more complex, while a chaotic series can become extremely hard.

The standard answer to this challenge in robotics is the *Auto-regressive (AR)* world model, which predicts one step ahead and feeds its prediction back as the next input (Chua et al., 2018; Janner et al., 2019; Gibson et al., 2023). With Temporal Convolutional Networks (TCN) in particular, this model is a common choice for learning mobile-robot dynamics (Saviolo et al., 2022; Bauersfeld et al., 2021). Its strength is its inductive bias: an AR model learns a single one-step map, a comparatively simple function that is reused at every step of the horizon, so every training transition informs every step of the rollout, making AR models data-efficient. Its weakness is the feedback loop itself: each prediction error becomes the next input, so an AR model pays the full cost of noise amplification. AR models therefore excel at the contracting end of the spectrum and degrade as sensitivity grows. The alternative is the *Multi-step (MS)* world model, which predicts the whole horizon in a single forward pass from the observed history, so its own predictions are never fed back as inputs. Asadi et al. (2019) show that this reduces the dominant term of the compounding-error bound from quadratic to linear in the horizon length. Their weakness is the mirror image of the AR strength: each single-shot MS forecast is a separate output, and nothing in the model ties the single-shot MS forecast map to the composition of one-step maps, so the structure shared across the horizon must be relearned from data rather than reused, making MS models comparatively data-hungry. Therefore, neither class dominates: we expect AR models to prevail at the contracting end of the spectrum, where compounding is benign, and their inductive bias pays off, and MS models to prevail at the chaotic end, where the exponential amplification of fed-back errors outweighs the cost of learning each step separately.

To address these limitations, we propose *Multi-Step Temporal Mixture Probabilistic (MTM-Pro)*, a hybrid world model that combines both classes: it is *trained* with an AR and an MS objective, and *deployed* as a one-step AR model. For each horizon step, the mixture function learns where to sit between the two models. Because both objectives train the same network, the next-step prediction used at deployment is shaped by the multi-step objective, which penalizes one-step predictions whose errors would compound. To demonstrate the resilience of our solution, we evaluate on five datasets chosen to span from low to high sensitivity to initial conditions. At the contracting end, two quadcopter datasets were recorded under a tracking controller: PI-TCN Easy (Saviolo et al., 2022), on nominal trajectories, and NeuroBEM (Bauersfeld et al., 2021), on more aggressive ones that the controller tracks less tightly. In the middle, we recorded an off-road uncrewed ground vehicle (UGV) dataset under open-loop commands held regardless of feedback, so deviations persist and accumulate.

Towards the sensitive end, PI-TCN Hard was recorded with the same quadcopter flying at the edge of its control envelope, where it is close to unstable. At the chaotic end, the Lorenz attractor turns any deviation into an exponentially diverging trajectory. Therefore, our contributions are:

- **A sensitivity view of compounding error.** We link the growth of compounding error to the modeled dynamic's sensitivity to initial conditions. This view predicts that AR and MS world models trade places along this spectrum.
- **A hybrid world model MTM-Pro.** This model is trained with coupled AR and MS objectives, whose overlapping multi-step forecasts are fused by a learned temporal mixture.
- **An open-loop off-road UGV dataset.** This dataset is released with this paper: 1.4 h of driving with a UGV on different terrains.This data differs from the quadcopter trajectories as it was recorded under open control loops, where path deviations are not corrected.
- **An empirical study** testing MTM-Pro against five standard AR and the two existing MS baselines in mobile robotics.

2 Related Work

Compounding error in learned dynamics. When a one-step predictor trained with teacher forcing is rolled out recursively, it receives its own inaccurate predictions as input — "fake" inputs the model was never trained on — causing compounding error (Lambert et al., 2019; Talvitie, 2017). Two complementary remedies exist (Moerland et al., 2023). The first is *single-shot non-autoregressive forecasting*: bypass the feedback loop entirely by emitting the whole window of future states in a single forward pass, so no previous prediction is ever fed back as input. Asadi et al. (2019) formalize this concept with the introduction of Multi-step Model for Model-based Reinforcement Learning (M3), a non-autoregressive multistep forecasting model conditioned on the true initial state and the full action sequence, reducing the dominant error coefficient from quadratic to linear in the forecast horizon length; following this principle, direct multi-output models have been applied to trajectory-based prediction (Lambert et al., 2021), fully convolutional quadrotor dynamics (Looper & Waslander, 2022) with the introduction of End-to-End Temporal Convolutional Network (E2E-TCN), and simultaneous Stochastic Model Predictive Control (MPC) for mobile robots (Yang et al., 2022). The second remedy retains the AR structure but *abandons teacher forcing*. Therefore, the model's own predictions are fed back during training, and a loss is computed over the entire forecasting horizon, exposing the model to the distribution shift that Talvitie (2017) identifies as the operative bottleneck, which includes the hallucination approach (Talvitie, 2017; Venkatraman et al., 2015), multi-step losses (Janner et al., 2019), and initialized LSTM models with integrated loss for off-road UGVs (Gibson et al., 2023). Moreover, Somalwar et al. (2025) show theoretically that well-specified one-step models attain the lower asymptotic error while direct multi-step predictors reduce bias under misspecification, where neither paradigm dominates across the sensitivity spectrum. To our knowledge, MTM-Pro is the first model to learn a mixture of the two: each AR-step issues a single-shot multistep unroll, and the overlapping unrolls of successive AR-steps are fused under a whole-horizon loss, with weights that are learned rather than hand-tuned.

Inference speed and data efficiency. Diffusion and large-transformer world models (Hafner et al., 2025; Hansen et al., 2024) require orders of magnitude more data and compute than the minutes-to-hours of trajectories a real-robot budget allows. Prior direct multi-step models grow in parameters with the forecast horizon (Lin et al., 2025; Lambert et al., 2021; Somalwar et al., 2025), and physics-informed neural SDEs (Djeumou et al., 2023) rely on non-transferable physics priors. Moreover, low-latency TCN backbones (Bauersfeld et al., 2021; Saviolo et al., 2022) achieve real-time inference but only as one-step predictors (Lea et al., 2016; Saviolo et al., 2022). A world model for an autonomous robot must balance all of these limitations. MTM-Pro targets this regime with a lightweight architecture, small real-robot training budgets, and a single forward pass at deployment.

3 METHODOLOGY

3.1 MTM-PRO: AR REACTIVITY AND MULTISTEP DRIFT RESISTANCE IN ONE MODEL

Notation. We denote by D the feature dimension and by H, U, F the history, unroll and forecast horizon lengths respectively. Sequences are indexed one-based: $\tau \frown \tau'$ concatenates two finite sequences, brackets $[\cdot]$ denote sub-sequence indexing (e.g., $\tau[1]$ is the first element and $\tau[2{:}]$ drops it), and a subscript range $\tau_{a:b}$ denotes $\tau_a, \tau_{a+1}, \dots, \tau_b$ (both ends included, hence $b-a+1$ elements); index sets $[a:b]$ follow the same inclusive convention. A moment-pair sequence $\left(\mu_{1:n}, \sigma^2_{1:n}\right) \in \mathbb{R}^{n\times\mathsf{D}}\times\mathbb{R}^{n\times\mathsf{D}}$ is indexed and concatenated entry-wise, an entry being a pair in $\mathbb{R}^{\mathsf{D}}\times\mathbb{R}^{\mathsf{D}}$; $\mathbb{P}(\mathbb{R})$ denotes the set of probability distributions on $\mathbb{R}$, and a moment pair $\left(\mu, \sigma^2\right)$ is a distribution parameterization. We write it as a Gaussian $\mathcal{N}(\mu, \sigma^2)$ everywhere in the text for simplicity, but any distribution parameterized by a mean and a variance can take its place, and the experiments use the Laplace distribution for its robustness to noise property (Appendix A, Section E).

State, action, history and state forecast. At every time step t two distinct things happen: the sensors report a *state* $\mathbf{s}_t$, and the controller issues an *action* $\mathbf{a}_t$. The *history*, records the last H time steps of state-action pairs $(\mathbf{s}_t, \mathbf{a}_t) \in \mathbb{R}^{\mathsf{D}}$. The *observation features* has index set $\mathcal{O}$, so $\mathbf{s}_t \in \mathbb{R}^{|\mathcal{O}|}$ and the *action features* have index set $\mathcal{A}$, so $\mathbf{a}_t \in \mathbb{R}^{|\mathcal{A}|}$), with $\mathsf{D} = |\mathcal{O}| + |\mathcal{A}|$. The same entry layout is used for everything the model outputs or is compared against (unroll, forecast, target), so every object written with a horizon superscript below is a sequence of state-action pairs; only the un-superscripted $\mathbf{s}_t$ is a state alone. An entry also has a *probabilistic* form, a moment pair $\left(\mu_t, \sigma^2_t\right) \in \mathbb{R}^{\mathsf{D}} \times \mathbb{R}^{\mathsf{D}}$ of per-feature mean and variance, action features included. Moreover, a hat ($\hat{\mathbf{s}}$) marks a model prediction. A superscript denotes a sequence rolled out from the subscript timestep: *backward*, the history $\mathbf{s}^{\mathsf{H}}_t \coloneqq (\mathbf{s}, \mathbf{a})_{t-\mathsf{H}+1:t}$ is the sequence of the H realized entries up to t, and its probabilistic form, the *augmented history* $\bar{\mathbf{s}}^{\mathsf{H}}_t \in \mathbb{R}^{\mathsf{H}\times\mathsf{D}} \times \mathbb{R}^{\mathsf{H}\times\mathsf{D}}$, is what the model actually consumes, realized entries entering it with a near-zero variance $\epsilon_0 \ll 1$ since they are realizations, not predictions; *forward*, the *state forecast* $\mathbf{s}^{\mathsf{F}}_{t+1} \coloneqq (\mathbf{s}, \mathbf{a})_{t+1:t+\mathsf{F}}$ is the sequence of the F next entries, whose probabilistic form $\left(\mu_{t+1:t+\mathsf{F}}, \sigma^2_{t+1:t+\mathsf{F}}\right) \in \mathbb{R}^{\mathsf{F}\times\mathsf{D}} \times \mathbb{R}^{\mathsf{F}\times\mathsf{D}}$ is the *state forecast distribution* our model – using a simple MLP backbone – is trained to produce ($\mathbf{s}^{\mathsf{U}}_t$ likewise over U steps). When a loss needs one feature of a target entry we write $\mathbf{s}^{\mathsf{F}}_{t+1}[\,i\,]_d$, $d \in [1{:}\mathsf{D}]$: a feature of the state $\mathbf{s}_{t+i}$ if $d \in \mathcal{O}$, of the action $\mathbf{a}_{t+i}$ if $d \in \mathcal{A}$; likewise, a subscript $\mathcal{O}$ on an entry or a moment pair keeps its state features only, so that $\hat{\mathbf{s}}^{\mathsf{U}}_{t+1}[1]_{\mathcal{O}}$ is the predicted next state $\hat{\mathbf{s}}_{t+1}$.

Problem statement. We want one model, fed only with a history $\mathbf{s}^{\mathsf{H}}_t$, that keeps what makes AR forecasting practical, one forward pass per control step and no plan to commit to, while inheriting what makes multistep forecasting resistant to drift, a prediction issued several steps ahead is never fed back, and that says how uncertain it is about every step of the F-step horizon. Our approach is AR at train time but collapses to a single forward pass at deploy time: three components are chained at every AR-step until the horizon F is reached, the U-step model unrolls a probabilistic forecast $\hat{\mathbf{s}}^{\mathsf{U}}_{t+1}$ over U steps, the temporal mixture functions fuse the overlapping predictions of successive AR-steps into a single state forecast $\hat{\mathbf{s}}^{\mathsf{F}}_{t+1}$, and the history update function refreshes the history for the next AR-step.

U-step model. Where a probabilistic model (PM) outputs the mean and variance of the next state, our U-step model $\rho^d_\theta \colon \mathbb{R}^{\mathsf{H}\times\mathsf{D}}\times\mathbb{R}^{\mathsf{H}\times\mathsf{D}} \to \mathbb{R}^{\mathsf{U}\times\mathsf{D}}\times\mathbb{R}^{\mathsf{U}\times\mathsf{D}}$ maps a length-H augmented history to a length-U multistep prediction distribution, one moment pair per predicted entry. Therefore, state and action features are treated alike, as the action forecast is needed by the MS loop only, such that

$$\hat{\mathbf{s}}^{\mathsf{U}}_{t+1} \leftarrow \rho^d_\theta\left(\bar{\mathbf{s}}^{\mathsf{H}}_t\right). \tag{1}$$

The model is *distribution-based*, which means it is consuming $\bar{\mathbf{s}}^{\mathsf{H}}_t$ rather than a sampled history keeps the predicted uncertainty inside the AR loop instead of collapsing it to a sample at every step.

Temporal Mixture functions. Intuitively, at forecast step i the model holds up to U predictions of the same physical instant, issued at different *lead times*: the component from AR-step $k = i$ is the freshest, one-step-ahead prediction, exactly what an AR model feeds back, whereas the component from AR-step $k = i - \mathsf{U} + 1$ was issued $\mathsf{U} - 1$ steps earlier, from a history that had seen fewer of the

model's own predictions, and is what a single-shot multistep model would commit to. Mixing these components with learned weights lets the model decide, for each forecast step, how much to trust its freshest prediction against those issued earlier, and this trade-off between reacting and committing is exactly what makes MTM-Pro a hybrid of the two paradigms rather than either one. To keep track of who predicted what, the F successive unrolls are stacked in the *forecast-distribution buffer* $\widehat{\mathbf{F}} \in \mathbb{R}^{\mathsf{F}\times\mathsf{F}\times\mathsf{D}} \times \mathbb{R}^{\mathsf{F}\times\mathsf{F}\times\mathsf{D}}$, an $\mathsf{F}\times\mathsf{F}$ grid of moment pairs. Therefore, each AR-step k writes its unroll along row k, into entries k to $\min(k+\mathsf{U}-1, \mathsf{F})$, so that column $\widehat{\mathbf{F}}_i \in \mathbb{R}^{\mathsf{F}\times\mathsf{D}} \times \mathbb{R}^{\mathsf{F}\times\mathsf{D}}$ collects every prediction ever made for step i. The *mixture function* $\rho_\phi \colon \left(\mathbb{R}^{\mathsf{F}\times\mathsf{D}} \times \mathbb{R}^{\mathsf{F}\times\mathsf{D}}\right) \times \mathbb{Z} \to \mathbb{P}(\mathbb{R})^{\mathsf{D}}$ consumes the column $\widehat{\mathbf{F}}_i$ together with its step index i, and outputs, for each feature, the weighted mixture of all unroll components overlapping that step following

$$\rho_\phi\big(\widehat{\mathbf{F}}_i, i\big) = \sum_{k=\max(i-\mathsf{U}+1,\,1)}^{i} \alpha_{k,\phi}(i)\, \mathcal{N}\Big(\mu_i^{(k)},\, {\sigma_i^{(k)}}^2\Big), \tag{2}$$

where $\left(\mu_i^{(k)}, {\sigma_i^{(k)}}^2\right) \leftarrow \widehat{\mathbf{F}}_i^k$ is the prediction for step i issued by the unroll of AR-step k, the only entries of the column that hold a forecast for step i, and the *weight function* $\alpha \colon \mathbb{Z}\times\mathbb{Z} \to [0,1]$, $\alpha_{k,\phi}(i) \coloneqq e^{\phi_i^k} \,/\, \sum_{k'=\max(i-\mathsf{U}+1,\,1)}^{i} e^{\phi_i^{k'}}$, is a softmax over the components active at step i of the learned, unconstrained *mixture logits* $\phi \in \mathbb{R}^{\mathsf{F}\times\mathsf{F}}$. Moreover, the logits start at zero, so every component is trusted equally at first, and the data decides afterwards. Both are applied feature by feature, with component moments $\left(\mu_{i,d}^{(k)}, {\sigma_{i,d}^{(k)}}^2\right)$, so the state-forecast entry $\hat{\mathbf{s}}_{t+1}^{\mathsf{F}}[\,i\,] \leftarrow \rho_\phi\big(\widehat{\mathbf{F}}_i, i\big)$ is a per-feature mixture whose moments $\left(\mu_i, \sigma_i^2\right)$ are what the history update below feeds back.

History update function. After each AR-step, the history update function $h \colon \left(\mathbb{R}^{\mathsf{H}\times\mathsf{D}} \times \mathbb{R}^{\mathsf{H}\times\mathsf{D}}\right) \times \left(\mathbb{R}^{\mathsf{F}\times\mathsf{D}} \times \mathbb{R}^{\mathsf{F}\times\mathsf{D}}\right) \times \mathbb{Z} \to \mathbb{R}^{\mathsf{H}\times\mathsf{D}} \times \mathbb{R}^{\mathsf{H}\times\mathsf{D}}$ slides the augmented history forward by one and appends the moments of the freshly mixed forecast, such that

$$h\big(\bar{\mathbf{s}}_{t+i-1}^{\mathsf{H}}, \widehat{\mathbf{F}}_i, i\big) \;=\; \bar{\mathbf{s}}_{t+i-1}^{\mathsf{H}}[2{:}\,] \frown \big(\mu_i, \sigma_i^2\big), \quad \mu_i = \textstyle\sum_k \alpha_{k,\phi}(i)\, \mu_i^{(k)}, \;\; \sigma_i^2 = \sum_k \alpha_{k,\phi}(i)\, {\sigma_i^{(k)}}^2, \tag{3}$$

over the same components k as (2), so (μ, σ^2) pairs propagate through the history without a sample ever being drawn. The variance σ_i^2 is deliberately chosen to be the weighted average of the component variances alone. Moreover, the extra mixture variance, as when its components disagree on the mean, is left out because that disagreement reflects the model hesitating between its fresh and its older predictions rather than the uncertainty of the system dynamics. Feeding back this mixture variance would make the history noisier at every step.

3.2 MTM-Pro composite objective and the loss-balancing problem

MTM-Pro is trained on two likelihood terms sharing the parameters (θ, ϕ), both computed during the training phase only. Therefore, the *multistep forecast* loss (i.e., MS loss for short) scores the per-step mixtures $\hat{\mathbf{s}}_{t+1}^{\mathsf{F},\mathrm{MS}} \in \mathbb{P}(\mathbb{R})^{\mathsf{F}\times\mathsf{D}}$ collected by the MS loop, and the Compounded predictions (CP) loss the chained first-step moment pairs $\hat{\mathbf{s}}_{t+1}^{\mathsf{F},\mathrm{CP}} \in \mathbb{R}^{\mathsf{F}\times\mathsf{D}} \times \mathbb{R}^{\mathsf{F}\times\mathsf{D}}$ collected by the CP loop (first two rows of Table 2), both against the target $\mathbf{s}_{t+1}^{\mathsf{F}}$, entry by entry and feature by feature, $\mathbf{s}_{t+1}^{\mathsf{F}}[\,i\,]_d$, $d \in [1{:}\,\mathsf{D}]$, with the scalar density $p(x \mid \mu, \sigma^2) = \mathcal{N}(x; \mu, \sigma^2)$.

Multistep forecast loss (MS). Instead of collapsing the per-step forecast $\hat{\mathbf{s}}_{t+1}^{\mathsf{F},\mathrm{MS}}[\,i\,] = \rho_\phi\big(\widehat{\mathbf{F}}_i, i\big)$ to its moments (3), each feature is scored under its own mixture of overlapping unroll components, and the horizon is aggregated with a temporal discount $\gamma_{\mathrm{MS}} \in (0, 1]$,

$$\mathcal{L}_{\mathrm{MS}} = \frac{1}{\mathsf{D}} \sum_{d=1}^{\mathsf{D}} \sum_{i=1}^{\mathsf{F}} \gamma_{\mathrm{MS}}^{\,i-1}\, \ell_{i,d}^{\mathrm{MS}}, \qquad \ell_{i,d}^{\mathrm{MS}} = -\log \sum_{k=\max(i-\mathsf{U}+1,\,1)}^{i} \alpha_{k,\phi}(i)\, p\Big(\mathbf{s}_{t+1}^{\mathsf{F}}[\,i\,]_d \Big| \mu_{i,d}^{(k)}, {\sigma_{i,d}^{(k)}}^2\Big), \tag{4}$$

so the model is rewarded for placing at least one sharp component on the truth rather than for hedging with a single wide Gaussian, and the next step is always scored at full weight. This way, all D features of the entry are scored, action features included, since the MS loop feeds its own action forecast back (Appendix D).

Compounded-prediction loss (CP). A deployed world model chains its own next-state prediction F times, which is where compounded error arises (Janner et al., 2019), a regime the MS loss never sees. The CP loss closes this gap by differentiating end-to-end through that recursion (i.e., see CP loop of Table 2). At each step, the first predicted entry $\hat{\mathbf{s}}^{\mathsf{F},\mathrm{CP}}_{t+1}[\,i\,] = \big(\mu^{\mathrm{CP}}_i, \sigma^{\mathrm{CP}\,2}_i\big) \leftarrow \rho^d_\theta\big(\bar{\mathbf{s}}^{\mathsf{H}}_{t+i-1}\big)[1]$ is scored against the true state $\mathbf{s}_{t+i}$, then fed back as if it had been observed by the *observation update* h_{obs}, which appends a new history entry made of the predicted state mean, with the ϵ_0 variance since it is interpreted as a realisation, and of the dataset action $\mathbf{a}_{t+i}$ in the action features $\mathcal{A}$, an input the model does not control (Appendix D, Eq. (9)),

$$\mathcal{L}_{\mathrm{CP}} = \frac{1-\gamma_{\mathrm{CP}}}{1-\gamma^{\mathsf{F}}_{\mathrm{CP}}} \sum_{i=1}^{\mathsf{F}} \gamma^{i-1}_{\mathrm{CP}} \, \frac{1}{|\mathcal{O}|} \sum_{d\in\mathcal{O}} -\log p\big(\mathbf{s}^{\mathsf{F}}_{t+1}[\,i\,]_d \;\big|\; \mu^{\mathrm{CP}}_{i,d}, \sigma^{\mathrm{CP}\,2}_{i,d}\big), \tag{5}$$

where only the observation features $\mathcal{O}$, the state proper, are scored. Then, the action features of the predicted entry are discarded, the true action being supplied by the dataset. Its per-step weights sum to one, so γ_{CP} only spreads a unit budget over the chained steps, its $i=1$ term being the conventional one-step loss. The constant gain between this and the un-normalized MS weights is absorbed by the composite objective auto-weighting below.

Objective, training and deployment. The composite objective $\mathcal{L}_{\text{MTM-Pro}}$ sums the MS and CP likelihoods, the optional orientation residuals (Appendix C) and the bound regularizer $\mathcal{R}_{\mathrm{bound}}$, each likelihood and residual entering through an *auto-weighted slot* $\mathcal{L}^{\beta}_j = c_j\, e^{-\beta_j}\, w_j\, \mathcal{L}_j + \frac{1}{2}\beta_j$ whose learned coefficient β_j (Kendall et al., 2018) replaces static hand-tuned weights for terms that differ in scale, sign and convergence rate (Appendix B). Training runs two AR loops on each sampled window, the MS loop feeding back its own mixed moments with h and the CP loop chaining first-step means with the dataset action through h_{obs}, then updates θ, ϕ and β jointly; at deployment only the second recursion survives, so MTM-Pro is multistep when it learns but a single-forward-pass AR one-step model when it runs, as fast as a single-step model (Appendix D, Table 2).

4 Results

Throughout, *drift* denotes the compounded prediction error after 5 s of open-loop forecasting rollout (i.e., labelled D@5s). The comparison is made at equal capacity: every model is sized to about 2.2M trainable parameters on Lorenz and about 5.1M on the robot datasets, with depth held fixed and the hidden width adjusted to meet the budget, so the differences reported below are due to the training paradigm and the backbone, not to model size. As for the experimental setup, we evaluate on five datasets chosen to span the spectrum of sensitivity to initial conditions, from least to most sensitive: PI-TCN *easy*, NeuroBEM, Husky, PI-TCN *hard*, and Lorenz. PI-TCN (Saviolo et al., 2022) and NeuroBEM (Bauersfeld et al., 2021) are real quadcopter datasets in which a controller tracks aggressive trajectories. Since the errors on PI-TCN are bimodal, we split it into PI-TCN *easy* and PI-TCN *hard*. Husky was recorded for this work with a ground robot driving off-road on grass and gravel, under random velocity commands each held in open loop for a given interval. Finally, the Lorenz system, a three-dimensional model of atmospheric convection and the canonical example of deterministic chaos, is a synthetic dataset created entirely in simulation. Together, they test fast flight where a controller damps errors, slow off-road driving where open-loop errors accumulate, and chaos where they explode. All the technical details of those datasets can be found in Appendix G.

4.1 Overall motion model performances

Table 1 compares the drift of every model on the five datasets, together with its inference rate and training time. Three observations set up the rest of the section. First, the best AR model is TCN: it is the best or tied-best AR baseline on every dataset, which confirms the backbone choice made in mobile robotics (Saviolo et al., 2022; Bauersfeld et al., 2021). And similarly, the best MS model is E2E-TCN. Second, the two classes do not fail in the same place. On the datasets recorded under a tracking controller, where deviations stay bounded, the MS baselines (E2E-TCN, M3) trail AR TCN by a wide margin (e.g., 65 cm against 10 cm of drift on PI-TCN) at the price of learning every horizon step as a separate output from the same data. On the chaotic Lorenz system it is the AR baselines that break down: their fed-back errors are amplified exponentially and their drift intervals span several orders of magnitude, i.e., a fraction of the rollouts diverge altogether. Third, MTM-Pro

Table 1: Drift and Training-Time Comparison Across Datasets.

Type	Method	PI-TCN (Easy)		Neurobem		Husky		PI-TCN (Hard)		Lorenz	
		D@5s [cm]	Train [h]	D@5s [m]	Train [h]	D@5s [cm]	Train [h]	D@5s [cm]	Train [h]	D@5s [m]	Train [h]
Auto-Regressive	MLP	19^{+23}_{-9}	2.3	$1.1^{+2}_{-0.8}$	2.3	$\mathbf{5.5^{+19}_{-2.7}}$	0.3	$\underline{35}^{+4}_{-15}$	2.3	5.6^{+7}_{-3}	0.22
	GRU	94^{+16}_{-8}	4.3	$2.0^{+0.6}_{-0.7}$	4.3	$\mathbf{5.5^{+6.9}_{-3.1}}$	0.57	66^{+14}_{-24}	4.3	$\underline{4}^{+1}_{-2}$	0.33
	LSTM	87^{+24}_{-15}	4.5	$2.7^{+0.9}_{-0.8}$	4.5	$\mathbf{5.8^{+9.0}_{-2.9}}$	0.6	65^{+25}_{-12}	4.5	$6.8^{+0.7}_{-0.8}$	0.37
	MLP-Prob	40^{+18}_{-16}	4.3	NA	4.3	$\mathbf{5.9^{+19.8}_{-3.0}}$	0.62	$\underline{33}^{+6}_{-8}$	4.3	$81^{+1\times10^4}_{-52}$	0.46
	TCN	$\mathbf{7.9^{+2.3}_{-1.2}}$	23.0	$\underline{0.6}^{+0.7}_{-0.4}$	23	$\mathbf{4.6^{+16}_{-2.5}}$	2.6	$\underline{3.8}^{+2\times10^4}_{-3.6}$	23.0	82^{+213}_{-75}	0.57
MS	E2E-TCN	$\underline{82}^{+21}_{-15}$	10.0	$\underline{2.3}^{+3}_{-0.7}$	10.0	$\mathbf{8.5^{+9.8}_{-3.7}}$	0.22	$\underline{0.52}^{+0.20}_{-0.15}$	10.0	$\underline{4}^{+1}_{-1}$	0.79
	M3	159^{+45}_{-44}	9.6	5.4^{+2}_{-2}	9.6	$\mathbf{4.9^{+29}_{-2.3}}$	0.18	$1.5^{+1.0}_{-0.5}$	9.6	23^{+22}_{-17}	0.89
Ours	MTM-Pro	$\mathbf{8.5^{+1.5}_{-1.1}}$	3.6	$\mathbf{0.07^{+0.1}_{-0.04}}$	3.6	$\mathbf{5.6^{+6.8}_{-3.0}}$	0.49	$\mathbf{0.08^{+0.2}_{-0.014}}$	3.6	$(\mathbf{7^{+1.5}_{-0.4}}) \times \mathbf{10^{-4}}$	0.28

Note: Drift values are reported as median$^{+(Q3-\text{median})}_{-(\text{median}-Q1)}$ ($Q1$/$Q3$: 25th/75th percentiles). *Abbreviations:* D@5s = Drift at 5 seconds; Train = Train Time; MS = Multi steps. **Bold** values indicate the best result in each column; underlined values indicate the best result in its class. The symbol NA means that the median, Q1, and Q3 have all diverged with values larger than 10^8.

is the best or tied-best model on all five datasets: on par with every model on the UGV dataset, where the slow dynamics barely separate the models, matching AR TCN on PI-TCN, and reducing drift several-fold on NeuroBEM and by more than four orders of magnitude on Lorenz. It does so at a cost comparable to the lightweight AR models, with CPU inference about four times faster and training about four times shorter than AR TCN (Section 4.4). The best class thus depends on the dataset, and the next subsection reads these results along the sensitivity spectrum to understand why.

4.2 Resilience across the sensitivity spectrum

As a recall, Section 1 ordered the datasets *a priori*, by how they were collected rather than by how the models fare on them: trajectories recorded under a tracking controller (PI-TCN Easy, NeuroBEM) are contracting, open-loop commands (UGV) let deviations persist and accumulate, flight at the edge of the control envelope (PI-TCN Hard) is close to unstable, and the Lorenz attractor amplifies any deviation exponentially. Fig. 2 places the drift distribution of the best model of each class, AR TCN and MS E2E-TCN, and of MTM-Pro along this ordering; the middle row reports the ratio of each baseline's drift to ours, so a ratio above one is a case where the baseline drifts more. The two baselines behave as the trade-off of Section 1 predicts. At the contracting end AR TCN is the more accurate: its one-step map is reused at every step and the controller present in the data keeps its fed-back errors bounded, whereas the MS model has to learn each horizon step from the same data and trails it by a factor 10 on PI-TCN Easy. As sensitivity grows the two curves cross: the gap narrows on the UGV and on PI-TCN Hard, and on Lorenz the MS model overtakes AR TCN, whose fed-back errors are now amplified exponentially. MTM-Pro follows neither curve: its ratio stays at or below one on every dataset, tied with AR TCN where compounding is benign and several-fold to orders of magnitude better where it is not. The point is not that

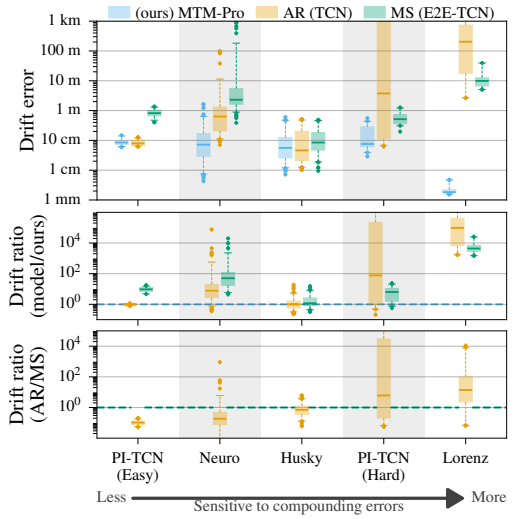


Figure 2: Drift after 5 s of the best AR model, the best multistep (MS) model and MTM-Pro per dataset, ordered *a priori* from least to most sensitive to initial conditions (left to right). Top: drift distributions. Middle: ratio of each baseline's drift to ours (dashed line at 1); a ratio above one means the baseline drifts more. Bottom: drift ratio of AR to MS; a ratio above one means that the AR drifts more than the MS model.

MTM-Pro wins each comparison by a wide margin, but that it is the only model whose long-horizon accuracy does not depend on which end of the spectrum the data comes from, which is what a world model deployed across regimes needs. The per-dataset drift-rate curves behind these snapshots are reported in Appendix I.

4.3 Training efficiency and data requirements

The argument of Section 1 makes a prediction about training, not only about accuracy: an AR model reuses one one-step map, so every transition informs every step of the rollout, whereas an MS model must learn each horizon step as a separate output and therefore needs more data, or more passes over it, to reach the same drift. Fig. 3 tests this prediction by tracking the drift of the best model of each class over the training epochs on PI-TCN, the real-robot dataset on which AR TCN comes closest to MTM-Pro in final accuracy (Table 1), so that convergence is compared where the outcome is tightest. MS E2E-TCN converges within a few tens of epochs but plateaus at a high drift, the signature of a model limited by the data at hand rather than by optimization. AR TCN keeps improving, but slowly and unsteadily: it needs about 150 epochs, some 19.5 h on our hardware, to become reliable, and only over the first 7 s of rollout, beyond which its drift never settles from one epoch to the next. MTM-Pro reaches its near-final drift in about 50 epochs, roughly 1.75 h, and the remaining epochs refine it without the instabilities of AR TCN: it inherits the reuse of the one-step map from its AR training while its MS objective supplies a whole-horizon signal from the first epoch. In practice, a model can be trained for a fraction of the epochs and still be accurate enough for long-horizon rollouts, which matters for robots, where trajectories are scarce and training budgets short.

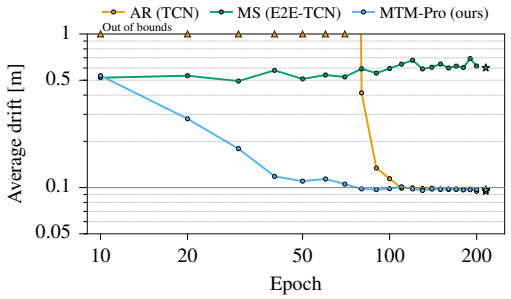


Figure 3: Average drift after 5 s over the training epochs for the best AR model, the best multistep (MS) model and MTM-Pro on the full PI-TCN dataset (8 seeds, 5 trajectories). A small triangle marks an average drift outside the y-axis bounds. MTM-Pro converges within 50 epochs, as fast as the MS model, and to a lower drift than the AR model.

4.4 Training time and inference frequency

Accuracy is only one axis of the choice a roboticist has to make: a world model must also run inside the control loop, at hundreds of forward passes per second for sampling-based MPC, and be retrained within the time budget of a field campaign. Fig. 4 plots the training time of every model against its inference rate on PI-TCN, on a CPU (circles) and a GPU (triangles), with the marker size growing as drift shrinks; Table 1 reports the same quantities on every dataset. Two groups emerge. The lightweight recurrent and feed-forward AR models (GRU, LSTM, MLP) are the cheapest, at 240 Hz to 290 Hz on CPU and 5 h to 8 h of training, but they are also the least accurate outside the UGV dataset. The convolutional models are the most accurate baselines, but also the most expensive to run and to train: AR TCN runs at about 48 Hz on CPU and trains for 53 h, the MS E2E-TCN at 32 Hz and 12 h (in mixed precision, Section E). MTM-Pro sits

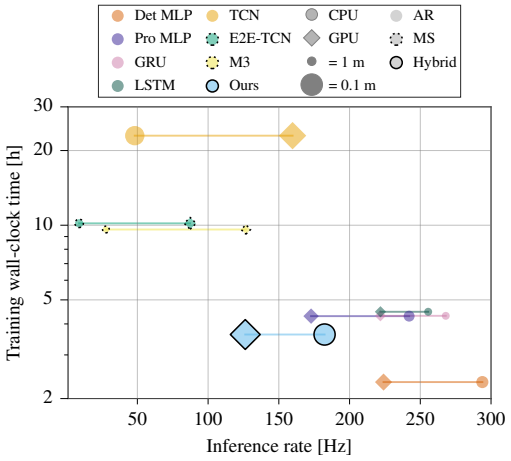


Figure 4: Training time against inference rate of every model on the full PI-TCN dataset, on CPU (circles) and GPU (triangles). Marker size is inversely proportional to the drift after 5 s: the larger the marker, the smaller the drift. MTM-Pro sits with the lightweight AR models in cost while carrying the lowest drift.

between the two groups, close to the lightweight one: about $186\,\mathrm{Hz}$ on CPU and $14\,\mathrm{h}$ of training on PI-TCN, i.e., roughly four times faster inference and four times shorter training than AR TCN for a lower drift, with a GPU rate comparable to both convolutional baselines. Per-device inference benchmarks on an embedded computer and the training wall-clock times measured on the HPC server are reported in Appendix I.

5 Conclusions

How fast a learned world model drifts is set as much by the dynamics of the system as by the model. The same one-step inaccuracy stays bounded under a tracking controller, accumulates under open-loop commands and explodes on a chaotic system. The two conventional ways of forecasting a horizon sit at opposite ends of this spectrum. AR models reuse a single one-step map, which makes them data-efficient but exposes them to the full amplification of their own fed-back errors; multistep models never feed back, but must learn every horizon step separately, which makes them data-hungry and slower to learn. We demonstrated that MTM-Pro has the best of both. Our world model is trained with an MS objective that fuses overlapping unrolls through a learned temporal mixture and a compounded-prediction objective that differentiates through the deployment recursion. Moreover, MTM-Pro is deployed as a one-step AR model, as light as a single-step predictor and without a plan to commit to. Across five datasets, including an open-loop off-road UGV dataset we release, ordered *a priori* by sensitivity to initial conditions, the two baseline classes traded places as predicted, while MTM-Pro was best or tied on every one, reducing drift several-fold on aggressive quadcopter flight and by orders of magnitude on the Lorenz attractor, with the inference rate and training time of a lightweight model and a convergence in a fraction of the epochs AR TCN requires. More broadly, our results suggest that a world model's reliability cannot be read from its one-step accuracy, nor from its success on a single regime: it should be evaluated on long-horizon drift across the sensitivity spectrum it will be deployed in, where the ranking of model classes can reverse. Moreover, we show indirectly that different datasets or applications exhibit different sensitivity to initial conditions. Deeper analysis of what type of trajectories a robot will produce, for example with an adaptation of the Lyapunov exponent to real-world noise, is still needed.

Limitations. Our evaluation measures open-loop drift, the regime the CP loss replicates; we have not closed the loop with a controller, nor evaluated the calibration of the horizon uncertainty the mixture provides, which we regard as the natural next step. The MS likelihood is a per-feature composite: it rewards multi-modality per channel but is blind to modes that couple several features, and it treats the horizon steps as independent marginals rather than as a joint trajectory density. The sensitivity ordering of datasets is argued from the collection protocol, not measured. The UGV is the only open-loop dataset and the only ground vehicle, so control regime and platform are confounded. Finally, MTM-Pro is a single network, so its epistemic uncertainty is carried by the temporal mixture alone, and we used a single unroll length $\mathsf{U} = \mathsf{F}$; the interplay between the unroll and forecast horizons is left to future work.

AI USE STATEMENT

In this work, we used generative AI tools to help implement our methods, to check the consistency of our derivations and proofs, and to critique our methodology and experiments as an additional internal reviewer. We did not use generative AI tools to formulate hypotheses, mathematical claims, or conceptual frameworks, to generate or clean data, or to interpret results.

On the software side, generative AI tools helped us prototype, refactor, and document our codebase, write unit and integration tests, and produce plotting code. They also helped convert hand-drawn sketches into diagrams and handwritten equations into LaTeX. Finally, we used them to improve the readability of the manuscript and to support the literature review by helping identify gaps in related work.

We have reviewed all AI-assisted work. For code, we first scoped each task against the current codebase and wrote a detailed implementation plan, then validated the generated code against that plan and our test suite. Converted equations and diagrams were checked against the originals, and literature suggestions were verified against the cited papers.

We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

In order for our work to be easily reproducible, we are releasing our project codebase on Github, including our training pipeline, baseline model, MTM-Pro model, and configuration file to reproduce our experiments. Reader who want to reproduce our work can refer to the Implementation details section of the Appendix. We are also linking all datasets we used for our experiments on the Github README.

Remark: we do not use fixed random seeds in our experiments. The reason why is that we are interested in the general stability of the model over a diversity of environments and seeding our experiment would insidiously mask the kind of behaviour we want to observe. So no seeds cherry-picking is possible and no chance of misinterpreting good results produce by accident because of a set of fixed-seed that happen to perform well.

ACKNOWLEDGMENTS

To preserve anonymity, acknowledgments will appear in the camera-ready version.

REFERENCES

Kavosh Asadi, Dipendra Misra, Seungchan Kim, and Michel L Littman. Combating the compounding-error problem with a multi-step model. *arXiv preprint arXiv:1905.13320*, 2019. URL https://arxiv.org/abs/1905.13320.

Dominic Baril, Simon-Pierre Deschênes, Luc Coupal, Cyril Goffin, Julien Lépine, Philippe Giguère, and François Pomerleau. DRIVE: Data-driven Robot Input Vector Exploration. In *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5829–5836, 2024.

Leonard Bauersfeld, Elia Kaufmann, Philipp Foehn, Sihao Sun, and Davide Scaramuzza. NeuroBEM: Hybrid Aerodynamic Quadrotor Model. In *Robotics: Science and Systems (RSS)*, 2021.

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2015. URL https://proceedings.neurips.cc/paper_files/paper/2015/file/e995f98d56967d946471af29d7bf99f1-Paper.pdf.

Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 4759–4770, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/3de568f8597b94bda53149c7d7f5958c-Abstract.html.

Franck Djeumou, Cyrus Neary, and Ufuk Topcu. How to Learn and Generalize From Three Minutes of Data: Physics-Constrained and Uncertainty-Aware Neural Stochastic Differential Equations. In *Conference on Robot Learning (CoRL)*, pp. 577–601, 2023. URL https://proceedings.mlr.press/v229/djeumou23a.html.

Andreas René Geist, Jonas Frey, Mikel Zhobro, Anna Levina, and Georg Martius. Learning with 3D rotations, a hitchhiker's guide to SO(3). In *International Conference on Machine Learning (ICML)*, volume 235, pp. 15331–15350, 2024.

Jason Gibson, Bogdan I. Vlahov, David D. Fan, Patrick Spieler, Daniel Pastor, Ali-akbar Aghamohammadi, and Evangelos A. Theodorou. A Multi-step Dynamics Modeling Framework For Autonomous Driving In Multiple Environments. In *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 7959–7965, 2023. URL https://doi.org/10.1109/ICRA48891.2023.10161330.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse control tasks through world models. *Nature*, 640(8059):647–653, 2025. URL https://www.nature.com/articles/s41586-025-08744-2.

Nicklas Hansen, Hao Su, and Xiaolong Wang. TD-MPC2: Scalable, Robust World Models for Continuous Control. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://openreview.net/forum?id=Oxh5CstDJU.

Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to Trust Your Model: Model-Based Policy Optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019. URL https://papers.nips.cc/paper_files/paper/2019/hash/5faf461eff3099671ad63c6f3f094f7f-Abstract.html.

Alex Kendall, Yarin Gal, and Roberto Cipolla. Multi-Task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7482–7491, 2018. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Kendall_Multi-Task_Learning_Using_CVPR_2018_paper.html.

Nathan Lambert, Daniel S. Drew, Joseph Yaconelli, Sergey Levine, Roberto Calandra, and Kristofer S. J. Pister. Low-Level Control of a Quadrotor With Deep Model-Based Reinforcement Learning. *IEEE Robotics and Automation Letters (RA-L)*, 4(4):4224–4230, 2019.

Nathan Lambert, Albert Wilcox, Howard Zhang, Kristofer S. J. Pister, and Roberto Calandra. Learning Accurate Long-term Dynamics for Model-based Reinforcement Learning. In *IEEE Conference on Decision and Control (CDC)*, pp. 2880–2887, 2021.

Colin S. Lea, René Vidal, Austin Reiter, and Gregory Hager. Temporal Convolutional Networks: A Unified Approach to Action Segmentation. In *ECCV Workshops*, pp. 47–54, 2016. URL https://doi.org/10.1007/978-3-319-49409-8_7.

Haoxin Lin, Yu-Yan Xu, Yihao Sun, Zhilong Zhang, Yi-Chen Li, Chengxing Jia, Junyin Ye, Jiaji Zhang, and Yang Yu. Any-step Dynamics Model Improves Future Predictions for Online and Offline Reinforcement Learning. In *International Conference on Learning Representations (ICLR)*, 2025.

Samuel Looper and Steven L. Waslander. Temporal Convolutions for Multi-Step Quadrotor Motion Prediction. In *IEEE Conference on Robots and Vision (CRV)*, pp. 32–39, 2022.

Edward N. Lorenz. Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences*, 20(2): 130–141, 1963. doi: 10.1175/1520-0469(1963)020⟨0130:DNF⟩2.0.CO;2.

Thomas M. Moerland, Joost Broekens, Aske Plaat, and Catholijn M. Jonker. Model-based Reinforcement Learning: A Survey. *Foundations and Trends in Machine Learning*, 16(1):1–118, 2023.

Alessandro Saviolo, Guanrui Li, and Giuseppe Loianno. Physics-Inspired Temporal Learning of Quadrotor Dynamics for Accurate Model Predictive Trajectory Tracking. *IEEE Robotics and Automation Letters (RA-L)*, 7(4):10256–10263, 2022.

Anne Somalwar, Bruce D. Lee, George J. Pappas, and Nikolai Matni. Learning with Imperfect Models: When Multi-step Prediction Mitigates Compounding Error. In *IEEE Conference on Decision and Control (CDC)*, pp. 82–89, 2025.

Erik Talvitie. Self-Correcting Models for Model-Based Reinforcement Learning. In *AAAI Conference on Artificial Intelligence*, 2017. URL https://ojs.aaai.org/index.php/AAAI/article/view/10850.

Arun Venkatraman, Martial Hebert, and J. Andrew Bagnell. Improving Multi-Step Prediction of Learned Time Series Models. In *AAAI Conference on Artificial Intelligence*, 2015. URL https://ojs.aaai.org/index.php/AAAI/article/view/9590.

Andong Yang, Wei Li, and Yu Hu. SMS-MPC: Adversarial Learning-based Simultaneous Prediction Control with Single Model for Mobile Robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 10905–10912, 2022.

A Notation and type conventions

The Methodology section (Section 3) define several objects that are chain into one another. This appendix makes explicit the small set of conventions that guarantees every one of these are well typed. Nothing here is needed to follow the main text; it is meant for the reader who wants to go further.

A moment pair is one probabilistic entry. Every probabilistic object of the paper is built from the same brick: a *moment pair* $(\mu, \sigma^2) \in \mathbb{R}^{\mathsf{D}} \times \mathbb{R}^{\mathsf{D}}$, one mean and one variance per feature of an entry $(\mathbf{s}_t, \mathbf{a}_t)$, that is for the state features and the action features alike (Section 3.1). A moment pair is therefore the probabilistic form of a whole entry, not of a plain state $\mathbf{s}_t \in \mathbb{R}^{|\mathcal{O}|}$, which never carries an action; the history $\mathbf{s}_t^{\mathsf{H}}$, the forecast $\mathbf{s}_{t+1}^{\mathsf{F}}$, the unroll $\mathbf{s}_t^{\mathsf{U}}$ and their predicted counterparts are all sequences of such entries. A *moment-pair sequence* $\left(\mu_{1:n}, \sigma^2_{1:n}\right) \in \mathbb{R}^{n \times \mathsf{D}} \times \mathbb{R}^{n \times \mathsf{D}}$ is simply n such bricks in a row, $\big((\mu_1, \sigma_1^2), \ldots, (\mu_n, \sigma_n^2)\big)$. Indexing $[\,\cdot\,]$ and concatenation $\frown$ therefore act on whole bricks: $\bar{\mathbf{s}}_t^{\mathsf{H}}[2{:}\,]$ drops the oldest (mean, variance) pair of the history, $\hat{\mathbf{s}}_{t+1}^{\mathsf{U}}[1]$ picks the first pair of an unroll, and $\bar{\mathbf{s}}_t^{\mathsf{H}}[2{:}\,] \frown \left(\mu, \sigma^2\right)$ appends a new pair at the end. This is all the history updates (3) and (9) do.

Reading the forecast-distribution buffer. The buffer $\widehat{\mathbf{F}}$ is an $\mathsf{F} \times \mathsf{F}$ grid whose cells are moment pairs. Rows are indexed by a superscript and columns by a subscript: row k is what AR-step k predicted, column i is everything ever predicted about forecast step i. Hence $\widehat{\mathbf{F}}^k_{a:b}$ is a slice of row k, written once by the unroll of AR-step k; $\widehat{\mathbf{F}}_i$ is column i, read once by the mixture function of step i; and the cell $\widehat{\mathbf{F}}^k_i \in \mathbb{R}^{\mathsf{D}} \times \mathbb{R}^{\mathsf{D}}$ is the prediction for step i issued at AR-step k. Since an unroll only covers U steps, only the cells with $k \le i \le \min(k + \mathsf{U} - 1, \mathsf{F})$ are ever written; the others keep their initial value and are never read.

From moment pairs to distributions. $\mathbb{P}(\mathbb{R})$ denotes the set of probability distributions on the real line, and $\mathbb{P}(\mathbb{R})^{\mathsf{D}}$ a tuple of D such distributions, one per feature. We write $\mathbb{P}(\mathbb{R})^{\mathsf{D}}$ rather than $\mathbb{P}(\mathbb{R}^{\mathsf{D}})$ because every distribution the model produces is factorized over features: it never models correlations between features. A moment pair is read as the distribution it parameterizes, $(\mu, \sigma^2) \mapsto \mathcal{N}(\mu, \sigma^2)$ feature by feature, so moment pairs are the special case of $\mathbb{P}(\mathbb{R})^{\mathsf{D}}$ in which each feature has a single component. The general case is needed in one place: the MS buffer $\hat{\mathbf{s}}_{t+1}^{\mathsf{F},\mathrm{MS}} \in \mathbb{P}(\mathbb{R})^{\mathsf{F} \times \mathsf{D}}$ holds, for each step and feature, the mixture (2) of up to U components, whereas the CP buffer $\hat{\mathbf{s}}_{t+1}^{\mathsf{F},\mathrm{CP}} \in \mathbb{R}^{\mathsf{F} \times \mathsf{D}} \times \mathbb{R}^{\mathsf{F} \times \mathsf{D}}$ holds one moment pair per step, the first entry of each unroll (1). Both are scored against the same target $\mathbf{s}_{t+1}^{\mathsf{F}}$, feature by feature, with the same scalar component density $p(x \mid \mu, \sigma^2) = \mathcal{N}(x; \mu, \sigma^2)$: the MS loss (4) evaluates a weighted sum of such densities over all D features of an entry, the CP loss (5) a single one over the state features $\mathcal{O}$ only.

What $\mathcal{N}$ stands for. Throughout the paper, $\mathcal{N}(\mu, \sigma^2)$ is a placeholder for whichever location–scale family is chosen, parameterized by its mean and log-variance. The derivations are written for the Gaussian for readability; the experiments use Laplace components with the same $(\mu, \log \sigma^2)$ head,

for the reasons given in Appendix E. Nothing in the recursions depends on this choice, since they only ever move means and variances around.

B Composite objective details

The composite objective (7) adds up terms of different nature: two log-likelihoods (MS and CP), possibly two squared residuals (orientation), and one regularizer. The main text gives the formulas; this appendix explains, term by term, why the constants are what they are. None of these constants is a contribution of this paper: they are the choices that make the terms comparable, so that the learned weighting of Kendall et al. (2018) can do its job. The slot constants c_j are justified where the auto-weighted slot (6) is introduced and are not discussed further here.

How the MS and CP losses spread their weight over the horizon. Both likelihood losses weight forecast step i by a discount $\gamma^{\,i-1}$, but they normalize it differently, and on purpose. The CP weights of (5) are divided by $\sum_{i=1}^{\mathsf{F}} \gamma_{\mathrm{CP}}^{i-1} = \frac{1-\gamma_{\mathrm{CP}}^{\mathsf{F}}}{1-\gamma_{\mathrm{CP}}}$, so they sum to one whatever γ_{CP} is. The discount then has a single, readable meaning: how a fixed budget of one is shared among the F chained steps. With γ_{CP} close to one the budget is spread almost evenly, so the late steps, where errors have compounded the most, count nearly as much as the first; with a small γ_{CP} the budget concentrates on the first few chained steps. Its first term ($i=1$) is exactly what a conventional one-step loss scores, so single-step supervision is never lost: it is the head of the CP sum. The MS weights of (4) are not divided by anything: step i is weighted by $\gamma_{\mathrm{MS}}^{i-1}$ as is. This has two convenient consequences. First, the next-step term ($i=1$) always has weight exactly one, whatever γ_{MS} and F are, so changing either hyperparameter never rescales the one-step supervision, it only decides how much the later steps add on top of it. Second, the weights $1, \gamma_{\mathrm{MS}}, \gamma_{\mathrm{MS}}^2, \ldots$ form a geometric series, so their sum never exceeds $1/(1-\gamma_{\mathrm{MS}})$: the MS loss cannot grow without bound when the horizon is lengthened. The price is a scale mismatch between the two losses. The MS weights sum to $Z_{\mathrm{MS}} = \sum_{i=1}^{\mathsf{F}} \gamma_{\mathrm{MS}}^{i-1}$ instead of one, so if the MS and CP losses scored every step equally badly, the MS loss would be Z_{MS} times the CP loss simply because of how they are normalized, not because of anything the model did.

What the static weights do, and how many coefficients are learned. The static weights w_j are hand-set, but they are not the loss balancing: they only fix the operating point at which the learned coefficients start, since every β_j is initialized at zero. Because they scale the data term but not the $\frac{1}{2}\beta_j$ regularizer, the model cannot make every term vanish at once by inflating every β_j: each unit of β_j costs $\frac{1}{2}$ regardless of w_j. Both orientation channels share the same static weight, $w_{\text{OR-}c} \coloneqq \lambda$. The number of learned coefficients follows the state: a velocity-only state (UGV) has $\mathcal{J}_{\mathrm{OR}} = \emptyset$ and learns two coefficients $(\beta_{\mathrm{MS}}, \beta_{\mathrm{CP}})$; a state with an heading (quadcopter) learns four.

Soft bounds on the predicted log-variance. A model that predicts its own variance has two easy ways to cheat: inflate the predicted variance on a hard sample so that a large error costs little, or shrink it to zero on an easy one so that the density blows up. Following Chua et al. (2018), every predicted log-variance ℓ is passed through a smooth two-sided squash with learnable bounds $\ell_{\min} < \ell_{\max}$,

$$\tilde{\ell} = \ell_{\max} - \mathrm{softplus}(\ell_{\max} - \ell), \qquad \ell^{\star} = \ell_{\min} + \mathrm{softplus}\Big(\tilde{\ell} - \ell_{\min}\Big).$$

The map is monotone, close to the identity when ℓ is well inside $[\ell_{\min}, \ell_{\max}]$, and only bends the tails, so it neither distorts ordinary predictions nor lets the variance escape. The bounds are trained with the network (we include them in θ). Left alone they would simply widen to whatever the data term prefers, so a linear penalty $\mathcal{R}_{\mathrm{bound}} = \kappa(\ell_{\max} - \ell_{\min})$, summed over the log-variance layers, is added once to the composite objective (7). It sits outside the learned weighting on purpose: no β_j can trade it off.

Automatic weighting of multiple loss terms. Summing terms that differ in scale and sign and converge at different rates, the likelihoods turning negative as the model sharpens while the CP term says little early in training but dominates later, defeats static hand-tuned weights, so we learn one unconstrained coefficient β_j per term, initialized at zero, following the uncertainty weighting of Kendall et al. (2018). Therefore, each term enters the objective as an *auto-weighted slot*, such that

$$\mathcal{L}_j^{\beta} \coloneqq c_j\, e^{-\beta_j}\, w_j\, \mathcal{L}_j + \tfrac{1}{2}\beta_j, \tag{6}$$

with w_j a static weight that only sets the initial operating point, and $c_j = 1$ for the likelihood terms, with $e^{-\beta_j}$ acting as an inverse temperature with log-normalizer $\frac{1}{2}\beta_j$, or $c_j = \frac{1}{2}$ for the orientation residuals, following the Kendall's Gaussian residual form. Then, the composite objective sums the active slots, following

$$\mathcal{L}_{\text{MTM-Pro}} = \sum_{j \in \mathcal{J}} \mathcal{L}_j^{\beta} + \mathcal{R}_{\text{bound}}, \qquad \mathcal{J} = \{\text{MS}, \text{CP}\} \cup \mathcal{J}_{\text{OR}}, \qquad \mathcal{J}_{\text{OR}} = \{\text{OR-MS}, \text{OR-CP}\}, \tag{7}$$

where $\mathcal{J}_{\text{OR}}$ is specialized orientation which we detail in Appendix C. We use this specialized orientation only when the state includes an orientation, and $\mathcal{R}_{\text{bound}} = \kappa(\ell_{\max} - \ell_{\min})$ penalizes the learnable soft log-variance bounds of Chua et al. (2018), added once outside the learned weighting (Appendix B).

C Orientation loss on $\mathrm{SO}(3)$

When the state carries an heading, e.g., the quadcopter's unit quaternion $\mathbf{q} \in S^3$, the per-feature likelihood is a poor heading error measure: $\mathbb{R}^4$ distances are not rotation angles and $\pm\mathbf{q}$ encode the same rotation. In this case we need a specialized orientation loss which is explained here.

We detail here the orientation loss $\mathcal{L}_{\text{OR-}c}$, $c \in \{\text{MS}, \text{CP}\}$, introduced in Section 3.2, and the two design choices behind it. The term only exists when the state contains an heading, as the quadcopter state does through a unit quaternion $\mathbf{q} = [w, \mathbf{v}] \in S^3$ (scalar first); on the velocity-only UGV state, $\mathcal{J}_{\text{OR}} = \emptyset$ and nothing below applies. We follow the recommendations of Geist et al. (2024) on learning with 3D rotations and recommend consulting this paper for more detail on distance metric for orientation.

Why a dedicated term. The per-feature likelihoods of (4) and (5) treat the four quaternion components as four ordinary real numbers. This is a poor error measure for a rotation: the Euclidean distance between two quaternions is not a rotation angle, and since $\mathbf{q}$ and $-\mathbf{q}$ encode the same rotation, two predictions can describe exactly the same heading while being as far apart as possible in $\mathbb{R}^4$. We keep the per-feature likelihood as the heading *uncertainty* model and add a term that measures the heading *mean* the geometric way, as a distance between rotations. It is also a safeguard against a well-known weakness of models that predict their own variance. In a likelihood loss, the penalty for an error in the mean is divided by the predicted variance, so the model has an easy escape route on the heading: when its predicted rotation is wrong, it can report a large variance and the wrong mean is barely penalized. The orientation term closes this escape route because it does not involve the variance at all: it measures only the distance between the predicted and the true rotation, so the mean is pulled toward the true rotation no matter how uncertain the model claims to be.

Definition. The network's raw four-vector for the heading block is normalized to unit length inside the computation graph, $\hat{\mathbf{q}} = \mathbf{v}/\|\mathbf{v}\|_2$, wherever a predicted heading leaves the model (unroll head and mixed forecast mean), so predicted headings are unit quaternions by construction. Let $R(\cdot) \in \mathrm{SO}(3)$ denote the rotation matrix of a unit quaternion, and $\hat{\mathbf{q}}_{t+i}^{c}$ the heading block of the predicted mean at forecast step i in channel c: the mean of the mixture $\hat{\mathbf{s}}_{t+1}^{\mathsf{F},\text{MS}}[\,i\,]$ for $c = \text{MS}$, the chained mean μ_i^{CP} for $c = \text{CP}$. The orientation loss is the squared *chordal* distance between predicted and true rotation matrices, averaged over the horizon,

$$\mathcal{L}_{\text{OR-}c} = \frac{1}{\mathsf{F}} \sum_{i=1}^{\mathsf{F}} \left\| R\big(\hat{\mathbf{q}}_{t+i}^{c}\big) - R(\mathbf{q}_{t+i}) \right\|_F^2 = \frac{8}{\mathsf{F}} \sum_{i=1}^{\mathsf{F}} \sin^2 \tfrac{\vartheta_{t+i}^{c}}{2}, \tag{8}$$

where $\vartheta_{t+i}^{c} \in [0, \pi]$ is the angle of the rotation that takes the predicted heading to the true one; the second equality is the standard chordal-to-angle identity (Geist et al., 2024). Three properties follow directly. The loss is *sign-invariant*: it only sees $R(\mathbf{q})$, and $R(\mathbf{q}) = R(-\mathbf{q})$, so no hemisphere flip or sign-continuity preprocessing is needed anywhere in the pipeline. It is horizon-averaged rather than summed, so it cannot grow with F and end up dominating the total loss at the expense of the likelihood terms. It is evaluated on physical (denormalized) headings, both quaternions being re-normalized before the map $\mathbf{q} \mapsto R(\mathbf{q})$.

Why chordal rather than geodesic. The natural alternative is the squared geodesic angle ϑ^2, the exact distance on $\mathrm{SO}(3)$. We use the chordal distance instead, following Geist et al. (2024): its gradient is well defined everywhere, whereas that of the geodesic angle is singular at $\vartheta = 0$, exactly where a converging model spends most of its time.

Two channels. The same loss is attached to the two training loops of Table 2, with separate learned weights. $\mathcal{L}_{\text{OR-MS}}$ scores the heading of the mixed forecast means of the MS loop; $\mathcal{L}_{\text{OR-CP}}$ scores that of the chained means of the CP loop, whose re-injected heading is normalized at each step exactly as at deployment. The CP channel is the one that matters most in practice: heading drift compounding along a free-running rollout is the first thing that breaks a controller, so it gets its own coefficient rather than being averaged into the MS term. Both channels enter the composite objective as auto-weighted slots (6) with $c_j = \frac{1}{2}$ and the shared static weight λ (Appendix B).

D Training and deployment details

This appendix expands the training and deployment summary of Section 3.2 with what was left out of the main text for space: the training iteration and the deployment recursion, the observation update h_{obs}, the training and deployment pseudo-code, and how the buffers are initialized.

Training and deployment. Each training iteration samples a trajectory $\tau \sim \mathcal{D}$ and a pivot t leaving room for a length-H history $\mathbf{s}_t^{\mathsf{H}}$ and a length-F target $\mathbf{s}_{t+1}^{\mathsf{F}}$, then runs the two AR loops that Table 2 contrasts with the deployment recursion. The pseudo-code is presented in algorithm 1 below as a complement. The MS loop mixes every unroll into $\hat{\mathbf{s}}_{t+1}^{\mathsf{F},\text{MS}}$ and feeds back the mixed moments of the whole entry with h, state and action features alike, so it never reads the dataset action. Only during a short warm-up is the MS loop scheduled-sampled (Bengio et al., 2015), with the true entry replacing the mixed forecast with a probability decaying exponentially to zero. The CP loop restarts from the observed window, keeps only the first unroll step in $\hat{\mathbf{s}}_{t+1}^{\mathsf{F},\text{CP}}$ and feeds back its state mean, paired with the dataset action, with h_{obs}, so gradients flow through the chained means exactly as errors compound at deployment. Both loops done, θ (log-variance bounds included), ϕ and β are jointly updated by gradient descent on $\mathcal{L}_{\text{MTM-Pro}}\left(\mathbf{s}_{t+1}^{\mathsf{F}} \middle| \hat{\mathbf{s}}_{t+1}^{\mathsf{F},\text{MS}}, \hat{\mathbf{s}}_{t+1}^{\mathsf{F},\text{CP}}\right)$ of (7). At deployment, MTM-Pro serves a downstream controller with a single forward pass (1) per control step: the controller consumes the state part of the first predicted entry $\hat{\mathbf{s}}_{t+1}^{\mathsf{U}}[1]$, the predicted next state $\hat{\mathbf{s}}_{t+1}$ (e.g., for MPC cost evaluation), the remaining unroll steps stay available for look-ahead, and h_{obs} advances the history with that predicted state and the controller's action. Neither the mixture function nor h is needed: together with the CP loss they shape the U-step model at train time only, so MTM-Pro is multistep when it learns and a pure AR one-step model when it runs, as fast as a single-step model. More details of the deployment are listed in the third row of Table 2 and in algorithm 2 below.

Table 2: The three recursions of MTM-Pro, all chaining the U-step model (1) through a sliding augmented history $\bar{\mathbf{s}}_t^{\mathsf{H}}$.

Recursion	**Kept from $\hat{\mathbf{s}}_{t+1}^{\mathsf{U}}$**	**Fed back into $\bar{\mathbf{s}}_t^{\mathsf{H}}$**	**Action source**	**Scored by**
MS loop (train)	all U steps, into $\widehat{\mathbf{F}}$	h (3): mixed moments (μ_i, σ_i^2)	own forecast, mixed action moments	$\mathcal{L}_{\text{MS}}$ (4) on $\hat{\mathbf{s}}_{t+1}^{\mathsf{F},\text{MS}}$
CP loop (train)	first step only	h_{obs} (9): mean with ϵ_0 variance	dataset $\mathbf{a}_{t+i}$	$\mathcal{L}_{\text{CP}}$ (5) on $\hat{\mathbf{s}}_{t+1}^{\mathsf{F},\text{CP}}$
Deployment	first step only	h_{obs} (9): mean with ϵ_0 variance	controller $\mathbf{a}_{t+1}$	none, sent to the controller

The one idea to keep in mind is that MTM-Pro chains the same U-step model in three places (Table 2), which differ only in what is written back into the history. The MS loop keeps all U steps of every unroll and feeds back, with h, the mixed forecast moments of the whole history entry, state and action features alike: it never reads the dataset action. The CP loop and the deployment keep only the first unroll step and feed it back with h_{obs}: the predicted state mean is written as if it had just been measured, with the ϵ_0 variance of an observation and its predicted variance discarded, while the

action is not predicted at all but taken from outside, the dataset action $\mathbf{a}_{t+i}$ in the CP loop and the controller's action at deployment. The CP loop is thus the training-time replica of the deployment recursion, restarted from the observed window and never teacher-forced, so that the CP loss trains the model in the very regime in which its errors compound.

The CP loop observation update function. At deployment the model only ever receives two things per control step: its own next-state prediction, and the action the controller has just chosen. The *observation update function* h_{obs} is the rule that writes these two things into the history so that the next forward pass can use them. It is the update of the deployment recursion, and the CP loop reuses it unchanged at train time; it is not the update of the MS loop, which uses h instead. To write it down we need to recall how a history entry is laid out (Section 3.1). A state and an action are two distinct things that happen at the same time step: the state is what the sensors report (velocities, heading, ...), the action is the command the controller sends to the actuators (motor thrusts, steering, ...). Once both have happened, the history records them side by side in one entry of D features: the state in the *observation features* and the action in the *action features*. We write $\mathcal{O}$ and $\mathcal{A}$ for the index sets of the two kinds of features, so that every feature index $d \in [1\!:\!\mathsf{D}]$ belongs to exactly one of them and $\mathsf{D} = |\mathcal{O}| + |\mathcal{A}|$ (Table 3); for a vector $v \in \mathbb{R}^{\mathsf{D}}$, v_d denotes its d-th feature. Given the current augmented history $\bar{\mathbf{s}}_t^{\mathsf{H}}$, a predicted next-entry moment pair $\left(\mu, \sigma^2\right)$ and the action $\mathbf{a} \in \mathbb{R}^{|\mathcal{A}|}$ of the incoming step,

$$h_{\text{obs}} \colon \left(\mathbb{R}^{\mathsf{H}\times\mathsf{D}} \times \mathbb{R}^{\mathsf{H}\times\mathsf{D}}\right) \times \left(\mathbb{R}^{\mathsf{D}} \times \mathbb{R}^{\mathsf{D}}\right) \times \mathbb{R}^{|\mathcal{A}|} \to \mathbb{R}^{\mathsf{H}\times\mathsf{D}} \times \mathbb{R}^{\mathsf{H}\times\mathsf{D}}$$

builds a new mean vector $\bar{\mu} \in \mathbb{R}^{\mathsf{D}}$ from the two inputs and appends it to the history,

$$h_{\text{obs}}\left(\bar{\mathbf{s}}_t^{\mathsf{H}}, \left(\mu, \sigma^2\right), \mathbf{a}\right) \;=\; \bar{\mathbf{s}}_t^{\mathsf{H}}[2\!:\!] \frown (\bar{\mu}, \mathbf{0}_{\mathsf{D}} + \epsilon_0), \qquad \bar{\mu}_d = \begin{cases} \mu_d & \text{if } d \in \mathcal{O} \text{ (predicted)}, \\ \mathbf{a}_d & \text{if } d \in \mathcal{A} \text{ (given)}, \end{cases} \tag{9}$$

where $\mathbf{a}_d$ denotes the entry of $\mathbf{a}$ that feature d stores. Read from left to right, the update does three things. First, $\bar{\mathbf{s}}_t^{\mathsf{H}}[2\!:\!]$ drops the oldest entry of the history, making room for one new entry. Second, the new entry is written with the near-zero variance ϵ_0 of an observation, exactly as a measured state would be: the predicted variance σ^2 is simply not used. Third, the mean of that new entry, $\bar{\mu}$, is assembled from two sources: its observation features are copied from the predicted mean μ, and its action features are filled with the given action $\mathbf{a}$. In other words, only the state is predicted, what the sensors would have measured, and that prediction is then treated as if it had just been measured; the action that goes with it in the entry is never predicted, it is an input the model does not control, handed to it by the dataset in the CP loop ($\mathbf{a}_{t+i}$) and by the controller at deployment. A deployed model has no other source of state, so this is the recursion in which errors compound, and the one the CP loss trains on. Unlike h, h_{obs} involves neither the mixture nor the forecast buffer, which is why it is the only update needed at deployment.

Buffer initialization. Each iteration draws a trajectory τ and a pivot t uniformly among $\{\mathsf{H}, \ldots, |\tau| - \mathsf{F}\}$, the timesteps that leave room for a length-H history $\mathbf{s}_t^{\mathsf{H}} = \tau_{t-\mathsf{H}+1:t}$ before them and a length-F target $\mathbf{s}_{t+1}^{\mathsf{F}} = \tau_{t+1:t+\mathsf{F}}$ after them. Two buffers collect what the loss will score, and their superscript names the loss they feed: $\hat{\mathbf{s}}_{t+1}^{\mathsf{F},\text{MS}} \in \mathbb{P}(\mathbb{R})^{\mathsf{F}\times\mathsf{D}}$ is initialized to a standard normal per entry, $\hat{\mathbf{s}}_{t+1}^{\mathsf{F},\text{CP}} \in \mathbb{R}^{\mathsf{F}\times\mathsf{D}} \times \mathbb{R}^{\mathsf{F}\times\mathsf{D}}$ to zero-mean, unit-variance moments. These initial values carry no information: every entry is overwritten before it is scored, so they only fix the type of each buffer (Notation and type conventions). The same holds for the forecast-distribution buffer $\widehat{\mathbf{F}}$ of the MS loop, whose unwritten cells are never read. The augmented history $\bar{\mathbf{s}}_t^{\mathsf{H}}$ is built from the observed window $\mathbf{s}_t^{\mathsf{H}}$ with the ϵ_0 variance, and rebuilt identically before the CP loop: both loops start from the same observed states, and the CP loop is never teacher-forced.

Training algorithm. algorithm 1 is the pseudo-code of the training iteration summarized above. Its two loops are the first two rows of Table 2: both chain the U-step model (1) from the same observed window, the MS loop through h (3) and the CP loop through h_{obs} (9), and a single gradient step is taken on the composite objective (7) once both are done.

Algorithm 1: MTM-Pro training phase

input: history length H, forecast length F, unroll length U, offline trajectory dataset $\mathcal{D}$
while *training* **do**
  $\tau \sim \mathcal{D}$
  $t \sim \mathcal{U}\{\mathsf{H}, \dots, |\tau| - \mathsf{F}\}$ ▶ *admissible pivot*
  $\mathbf{s}^{\mathsf{H}}_t \leftarrow \tau_{t-\mathsf{H}+1:t}$
  $\mathbf{s}^{\mathsf{F}}_{t+1} \leftarrow \tau_{t+1:t+\mathsf{F}}$
  $\hat{\mathbf{s}}^{\mathsf{F},\mathrm{MS}}_{t+1} \leftarrow \big(\mathcal{N}(0,1)\big)_{\mathsf{F}\times\mathsf{D}}$ ▶ *MS buffer* $\in \mathbb{P}(\mathbb{R})^{\mathsf{F}\times\mathsf{D}}$, *scored by* (4)
  $\hat{\mathbf{s}}^{\mathsf{F},\mathrm{CP}}_{t+1} \leftarrow (0_{\mathsf{F}\times\mathsf{D}}, 1_{\mathsf{F}\times\mathsf{D}})$ ▶ *CP buffer* $\in \mathbb{R}^{\mathsf{F}\times\mathsf{D}} \times \mathbb{R}^{\mathsf{F}\times\mathsf{D}}$, *scored by* (5)
  $\bar{\mathbf{s}}^{\mathsf{H}}_t \leftarrow \big(\mathbf{s}^{\mathsf{H}}_t, 0_{\mathsf{H}\times\mathsf{D}} + \epsilon_0\big)$
  $\widehat{\mathbf{F}} \leftarrow (0_{\mathsf{F}\times\mathsf{F}\times\mathsf{D}}, 1_{\mathsf{F}\times\mathsf{F}\times\mathsf{D}})$
  for $i \in [1:\mathsf{F}]$ **do** ▶ *MS loop*
    $\hat{\mathbf{s}}^{\mathsf{U}}_{t+i} \leftarrow \rho^d_\theta\big(\bar{\mathbf{s}}^{\mathsf{H}}_{t+i-1}\big)$ ▶ (1)
    $\widehat{\mathbf{F}}^{\,i}_{i:\min(i+\mathsf{U}-1,\mathsf{F})} \leftarrow \hat{\mathbf{s}}^{\mathsf{U}}_{t+i}\big[1:\min(\mathsf{U}, \mathsf{F}-i+1)\big]$ ▶ *row* i, *steps past* F *dropped*
    $\hat{\mathbf{s}}^{\mathsf{F},\mathrm{MS}}_{t+1}[\,i\,] \leftarrow \rho_\phi\big(\widehat{\mathbf{F}}_i, i\big)$ ▶ *mix column* i, (2)
    $\bar{\mathbf{s}}^{\mathsf{H}}_{t+i} \leftarrow h\big(\bar{\mathbf{s}}^{\mathsf{H}}_{t+i-1}, \widehat{\mathbf{F}}_i, i\big)$ ▶ (3)
  $\bar{\mathbf{s}}^{\mathsf{H}}_t \leftarrow \big(\mathbf{s}^{\mathsf{H}}_t, 0_{\mathsf{H}\times\mathsf{D}} + \epsilon_0\big)$ ▶ *reset, no teacher forcing*
  for $i \in [1:\mathsf{F}]$ **do** ▶ *CP loop, as algorithm 2*
    $\hat{\mathbf{s}}^{\mathsf{U}}_{t+i} \leftarrow \rho^d_\theta\big(\bar{\mathbf{s}}^{\mathsf{H}}_{t+i-1}\big)$ ▶ (1)
    $\hat{\mathbf{s}}^{\mathsf{F},\mathrm{CP}}_{t+1}[\,i\,] \leftarrow \hat{\mathbf{s}}^{\mathsf{U}}_{t+i}[1]$ ▶ *first predicted entry, moment pair*
    $\bar{\mathbf{s}}^{\mathsf{H}}_{t+i} \leftarrow h_{\mathrm{obs}}\big(\bar{\mathbf{s}}^{\mathsf{H}}_{t+i-1}, \hat{\mathbf{s}}^{\mathsf{F},\mathrm{CP}}_{t+1}[\,i\,], \mathbf{a}_{t+i}\big)$ ▶ (9), *dataset action*
  $\theta, \phi, \beta \leftarrow \theta, \phi, \beta - \nabla_{\theta,\phi,\beta}\, \mathcal{L}_{\text{MTM-Pro}}\Big(\mathbf{s}^{\mathsf{F}}_{t+1} \Big| \hat{\mathbf{s}}^{\mathsf{F},\mathrm{MS}}_{t+1}, \hat{\mathbf{s}}^{\mathsf{F},\mathrm{CP}}_{t+1}\Big)$ ▶ (7)
return: θ, ϕ, β

Deployment algorithm. algorithm 2 is the deployment recursion of the third row of Table 2. The augmented history is built once from the latest H observed states and the actions taken alongside them, paired with the ϵ_0 variance. Then, at every control step, the U-step model runs a single forward pass (1), the state part of its first predicted entry $\hat{\mathbf{s}}^{\mathsf{U}}_{t+1}[1]$, the predicted next state $\hat{\mathbf{s}}_{t+1}$, is sent to the controller, and the history is advanced with h_{obs} (9) fed with that entry and the action the controller chose; the action features of the predicted entry are ignored, since h_{obs} only reads its state features. As written, this is the *open-loop* rollout, in which the model runs on its own predictions: it is the recursion the CP loop replicates at train time and the regime our drift metric measures (Section 4). In a closed control loop, whenever a fresh observation is available it simply takes the place of the prediction in h_{obs}; nothing else changes.

Algorithm 2: MTM-Pro deployment phase

input: history length H, unroll length U, trained parameters θ
$\bar{\mathbf{s}}^{\mathsf{H}}_t \leftarrow \big(\mathbf{s}^{\mathsf{H}}_t, 0_{\mathsf{H}\times\mathsf{D}} + \epsilon_0\big)$ ▶ *latest* H *observed states and actions at start time* t
while *true* **do**
  $\hat{\mathbf{s}}^{\mathsf{U}}_{t+1} \leftarrow \rho^d_\theta\big(\bar{\mathbf{s}}^{\mathsf{H}}_t\big)$ ▶ *unroll-model* (1)
  $\hat{\mathbf{s}}_{t+1} \leftarrow \hat{\mathbf{s}}^{\mathsf{U}}_{t+1}[1]_{\mathcal{O}}$ ▶ *state part of the first predicted entry, sent to the downstream controller*
  $\bar{\mathbf{s}}^{\mathsf{H}}_{t+1} \leftarrow h_{\mathrm{obs}}\big(\bar{\mathbf{s}}^{\mathsf{H}}_t, \hat{\mathbf{s}}^{\mathsf{U}}_{t+1}[1], \mathbf{a}_{t+1}\big)$ ▶ (9), *controller action*
  $t \leftarrow t+1$

E Implementation details

This appendix collects the implementation choices shared by all experiments, in the order a reader re-implementing MTM-Pro would meet them: the distribution family, the practical consequences of the sampling-free recursion, the optimization schedule, the arithmetic precision and execution of the training step, and the hyperparameters. None of them is a contribution of this paper.

Distribution family: Laplace instead of Gaussian. MTM-Pro is not tied to a particular distribution: any family parameterized by a mean and a log-variance can be plugged into the same $(\mu, \log \sigma^2)$ head, since the recursions only ever move means and variances around. Both the Gaussian and the Laplace are implemented. Section 3 is written with the Gaussian for simplicity; the experiments use the Laplace because of its heavier tails: it treats a large deviation as plausible rather than exceptional, which makes the fit robust to outliers and therefore less affected by sensor noise and signal spikes in the recorded trajectories. This is a robustness choice driven by the datasets, mobile robots subject to adverse phenomena, and is unrelated to MTM-Pro itself.

Consequences of the sampling-free recursion. Because the history update h (3) moves means and variances rather than samples, three things follow. First, the forward pass is a deterministic function of the parameters: the MS objective carries no reparameterization or score-function noise, which is what lets a horizon of $\mathsf{F} = 10$ steps with $\mathsf{U} = 10$ overlapping unrolls train stably with a single network, without ensemble averaging. Second, uncertainty is an input, not only an output: the augmented history carries a mean and a log-variance per feature, so the network learns to propagate a belief, the discrete-time analogue of the predict step of a filter, rather than a state-to-state map. Realized entries, observed state and applied action alike, enter it with the certainty marker $\epsilon_0 = 10^{-3}$ ($\log \epsilon_0 \approx -6.9$), at ingestion, on scheduled-sampling teacher-forcing steps and in the observation update (9) alike. Third, the cost is F evaluations of the U-step model per training sample, with no Monte-Carlo multiplier. Two normalizations complete the picture: every predicted log-variance passes through the learnable soft bounds of Appendix B, initialized to $[\ell_{\min}, \ell_{\max}] = [-10, 0.5]$, and forecast moments, histories and targets all live in a single normalized space, a per-feature z-score with a symmetric standard deviation.

Optimization. The MS loop can be optionally teacher-forced for a scheduled period (Bengio et al., 2015) (Appendix D). We do it for very short warm-up period in our experiments Table 3. At each AR-step a coin decides whether the true entry (state and action), paired with the EPS variance, replaces the mixed forecast in the history; the probability of that replacement decays exponentially from one to zero over the first t_1 optimizer steps (Table 3), after which the MS loop is fully free-running. The CP loop, the evaluation and the deployment paths are never teacher-forced, so the schedule can only close the train/deploy gap, never widen it. All learned weighting quantities start neutral: the coefficients β_j are initialized to zero, so every slot of (6) starts with a unit multiplier $e^{-\beta_j} = 1$ and the static weights w_j alone set the initial operating point; the mixture logits ϕ are also zero-initialized, so every unroll component is trusted equally at first.

Hyperparameters – Optimizer and data level. Table 3 lists the loss-relevant constants of the Lorenz attractor and of the three robot arms. Every mechanism of the objective is the same code path with different constants; the only structural difference between the columns is the pair of orientation slots, present on the quadcopter state only. The number of composite loss weights $\#\{\beta_j\}$ follows directly from it: one learned coefficient per slot of (6), that is MS and CP everywhere, plus OR-MS and OR-CP when the orientation slots are present. On the Lorenz attractor the state holds the three coordinate increments of the attractor and we feed the time step Δt between two samples to the single action feature dimension; teacher forcing is not used at all on this dataset. One row deserves a comment. The CP discount is close to one on Lorenz, NeuroBEM/PI-TCN, so the CP budget is spread almost uniformly over the deploy horizon; it is much smaller on the UGV, concentrating the budget on the first chained steps (Appendix B).

Hyperparameters – Model-level. Table 4 completes Table 3 with the model-level settings of every model in our experiments: its backbone, whether its output is a point estimate or a distribution, its depth, its width and the arithmetic precision it was trained in. **The guiding rule is a fixed capacity per dataset: every model is sized to about 2.2M trainable parameters on the Lorenz attractor and about 5.1M on the UGV and the two UAV datasets, so that the comparison of Section 4 is between training paradigms and backbones at equal capacity, not between model sizes.** Depth is held fixed within a dataset and the hidden width is the free variable that brings each model onto the budget, which is why a probabilistic head (two outputs per feature instead of one) or a recurrent cell (three or four gates per unit) comes with a narrower layer. The two UAV datasets, NeuroBEM and PI-TCN, use the same settings; the UGV differs from them only for the two MS baselines, whose direct multi-output architecture depends on the horizon they are trained to forecast in one shot

Table 3: Loss-relevant hyperparameters common to all models on the Lorenz attractor and the three robot datasets.

Quantity	Symbol	Lorenz	NeuroBEM/PI-TCN	UGV
history / forecast / unroll lengths	$(\mathsf{H}, \mathsf{F}, \mathsf{U})$	$(20, 10, 10)$	$(20, 10, 10)$	$(20, 10, 10)$
feature width (obs + act)	$\mathsf{D} = \lvert\mathcal{O}\rvert + \lvert\mathcal{A}\rvert$	$3 + 1$	$10 + 5$	$6 + 3$
minibatch size	B	300	512	300
MS discount / static weight	$\gamma_{\mathrm{MS}}, w_{\mathrm{MS}}$	0.933, 1.0	0.83, 1.0	0.76, 1.0
CP discount / static weight	$\gamma_{\mathrm{CP}}, w_{\mathrm{CP}}$	0.933, 87.0	0.988, 30.0	0.74, 78.85
orientation static weight	λ	—	68.9	—
number of composite loss weights	$\#\{\beta_j\}$	2	4	2
MS teacher-forcing scheduled window (optimizer steps)	t_1	— (off)	716 (≈ 0.9 epoch)	252 (≈ 2.2 epochs)
log-variance bounds at initialization	$[\ell_{\min}, \ell_{\max}]$	$[-10, 0.5]$	$[-10, 0.5]$	$[-10, 0.5]$
certainty marker (observation variance)	ϵ_0	10^{-3}	10^{-3}	10^{-3}

($\mathsf{F} = 50$ on the UGV, 500 on the UAVs and 1000 on Lorenz), whereas every other model is trained on the lengths of Table 3.

The remaining choices are common to all models and datasets. Every hidden layer uses the GELU activation, including the E2E-TCN, whose original recipe (Looper & Waslander, 2022) uses ReLU, and no dropout. Each model is a single network, without ensemble. The feed-forward bodies, the MLPs, M3 and the encoder of MTM-Pro, are stacks of identity-preserving residual blocks $y = x + W_2\,\mathrm{GELU}(W_1 x)$ whose last layer is zero-initialized, so that every block starts as the identity; the output heads are plain dense layers.

AR TCN stacks causal convolutions of kernel size 3; the E2E-TCN follows the published architecture, causal convolutions of kernel size 5 with exponentially growing dilations (1 to 2^3 on the UGV, to 2^6 on the UAVs and to 2^9 on Lorenz, so that the receptive field covers the history plus the horizon), batch normalization and a residual skip every other block, and its last block projects directly onto the forecast, hence no separate decoder.

M3 (Asadi et al., 2019) exist in two variant, the first one shares one head across the horizon and conditions it on the step index and the second one instantiate one head per step, which at $\mathsf{F} = 500$ would exceed our server memory allocation in one training run just on experimental checkpointing. For that reason, whe pick M3 with one shares head variant. The AR recurrent models and AR TCN decode the last hidden state with one shared linear layer followed by the single-step head; on Lorenz that head is wider than the recurrent state (width in parentheses in Table 4). Every probabilistic output is a Laplace distribution parameterized by a mean and a log-variance per feature (Section E); for MTM-Pro it is the per-step mixture of the U overlapping unrolls of Section 3. All models are trained with AdamW and a per-epoch exponential learning-rate decay, on the same history length $\mathsf{H} = 20$ and the same per-feature z-score normalization.

Finally, the precision column reads as follows: "fp32" is plain 32-bit floating point with the TF32 shortcut disabled, and "bf16" is mixed precision, the forward pass and the loss under a `bfloat16` autocast with 32-bit master weights, optimizer state and backward pass. Only three rows read bf16: the two MS baselines, E2E-TCN and M3, and the AR TCN-bf16 control, which is the AR TCN row retrained under the same autocast. This is a deliberate choice, made for time, and it is checked rather than assumed; Section E states why and how.

Arithmetic precision. All MTM-Pro reported runs use plain 32-bit floating point from end to end. We also disable the TF32 shortcut that PyTorch may apply to 32-bit matrix products on recent GPUs, which reduces the precision of their inputs, and request the highest available matrix-multiplication precision instead (`torch.set_float32_matmul_precision("highest")`, the PyTorch v2.14 default). The recursion of MTM-Pro chains F forward passes and feeds predicted means and variances back into its own input, so we did not want rounding errors to have a second way of compounding along the horizon. Mixed-precision training (automatic casting of the linear layers to

Table 4: Model-level architecture settings of every model, per dataset. L_{enc}: number of encoder (body) layers, MLP blocks, recurrent layers or convolution blocks acting on the history; L_{dec}: number of decoder (head) layers producing the output; h: hidden width, with the decoder width in parentheses when it differs. All models have about 2.2M trainable parameters on Lorenz and about 5.1M on the UGV and UAV datasets. The two UAV datasets (NeuroBEM, PI-TCN) share the same settings; where the UGV differs from them, the cell reads UGV / UAV.

Model	**Backbone**	**Output**	**Lorenz**			**UGV / UAV**			**Precision**
			L_{enc}	L_{dec}	h	L_{enc}	L_{dec}	h	
(AR) Det MLP	MLP	deterministic	3	3	654	3	2	1088	fp32
(AR) Prob MLP	MLP	probabilistic, Laplace	3	3	519	3	2	899	fp32
(AR) GRU	recurrent, GRU	deterministic	2	3	377 (598)	3	2	544	fp32
(AR) LSTM	recurrent, LSTM	deterministic	3	3	260 (598)	3	2	476 (544)	fp32
(AR) TCN	causal convolution, $k = 3$	deterministic	2	3	377 (598)	3	2	544	fp32
(AR) TCN-bf16	causal convolution, $k = 3$	deterministic	2	3	377 (598)	3	2	544	bf16
(MS) E2E-TCN	dilated causal convolution, $k = 5$	deterministic	3	—	152	3	—	379 / 279	bf16
(MS) M3	MLP, horizon-indexed head	deterministic	3	2	1063	3	2	2178 / 1326	bf16
MTM-Pro-MS (ours)	MLP, residual encoder	probabilistic, Laplace mixture	3	3	456	3	2	744	fp32
MTM-Pro-MS+CP (ours)	MLP, residual encoder	probabilistic, Laplace mixture	3	3	456	3	2	744	fp32

`bfloat16` or `float16`) is implemented but was left off for MTM-Pro and for the AR baselines for the same reason, and because our profiling indicates it would have bought little for MTM-Pro: its training step is made of many small operations, one per unroll step on a state of a handful of features, and its wall-clock time follows the number of those operations rather than the size of the tensors they act on, which is what mixed precision reduces. See PyTorch documentation on Numerical accuracy for detail.

Why the MS baselines are trained in mixed precision, and how we control for it. The two MS baselines, E2E-TCN and M3, are the only models of Table 1 trained in mixed precision, **a choice dictated by the training budget rather than by accuracy.** Unlike MTM-Pro and the AR models, which forecast one step at a time on a low-dimensional state, the MS baselines produce the whole deploy horizon in one shot ($\mathsf{F} = 500$ steps on the UAV datasets, 1000 on Lorenz) through wide layers (h up to 2178 for M3, Table 4) or long dilated convolutions. Their training step is therefore dominated by a few large tensor operations, precisely the regime in which mixed precision is effective. We quantified the saving by benchmarking the training step of every model of Table 1 on the same A100 server, with the same batch size, training budget (400 epochs of 631 batches) and PI-TCN data path. For AR TCN, one step takes 326 ms in fp32 with TF32 disabled (23.6 h per run) and 72 ms under the `bfloat16` autocast (5.3 h per run), a measured speed-up of $4.5\times$. Under the same autocast, one E2E-TCN step takes 140 ms and one M3 step 130 ms, i.e. 10.2 h and 9.6 h per run, consistent with Table 1. Applying the AR TCN speed-up factor, a single fp32 run would take about 46 h for E2E-TCN and 43 h for M3, roughly 180 h of GPU time for one seed of both baselines on both UAV datasets, and several weeks of GPU time once repeated over the seeds of Table 1. This estimate is an extrapolation that assumes a comparable speed-up for E2E-TCN and M3; the assumption is

conservative, since their training steps are the most dominated by large tensor operations. That budget did not fit between the final experimental protocol and the submission, so the MS baselines, and only they, were trained under the autocast. **The AR TCN-bf16 row controls for this choice.** One could object that the MS baselines trail AR TCN on the contracting datasets (Section 4.1) because of their lower training precision rather than their training paradigm. To rule this out, we retrained the best AR model, TCN, under the same `bfloat16` autocast, with the same 32-bit master weights, optimizer state and backward pass, and with architecture, hyperparameters and seeds unchanged. If mixed precision were the cause of the gap, AR TCN-bf16 would drift substantially more than AR TCN; it does not, and in general performs slightly better. Conversely, mixed precision offered little to MTM-Pro: its training step consists of many small operations rather than a few large ones, so the autocast would have yielded a marginal speed-up while giving rounding errors a second way of compounding along the recursion. We therefore trained it in fp32 and addressed its cost by other means, as follows.

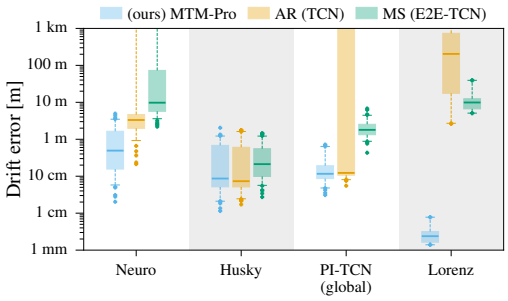


Figure 5: Drift after 5 s of the best AR model, the best multistep (MS) model and MTM-Pro per dataset, ordered *a priori*.

Training-step execution. The lever that does address the cost of MTM-Pro is a CUDA-graph captured training step: the whole "loss, backward pass, optimizer update" sequence is recorded once and replayed as a single unit at every step, with static input buffers. This changes only the wall-clock time, the loss, gradients and optimizer state are bit-identical to the eager step; in the training-step benchmark above, it makes a training step of MTM-Pro about 3.7 times faster (from 181 ms to 49 ms), whereas AR TCN barely moves (from 326 ms to 317 ms), which is consistent with where each model spends its time.

F PI-TCN SPLIT ANALYSIS

This appendix presents the analysis that led to splitting the PI-TCN dataset into easy and hard splits. The PI-TCN split comes from an analysis of the residuals of the best-performing baseline, TCN. Fig. 5 shows the drift of the best Autoregressive (AR), Multi-step (MS), and our proposed model over a 10-second horizon. While TCN is the best baseline on PI-TCN (global) by median, its third quantile shows a drift of 3.22×10^{12}, meaning PI-TCN could contain sub-datasets with two levels of motion-modeling difficulty.

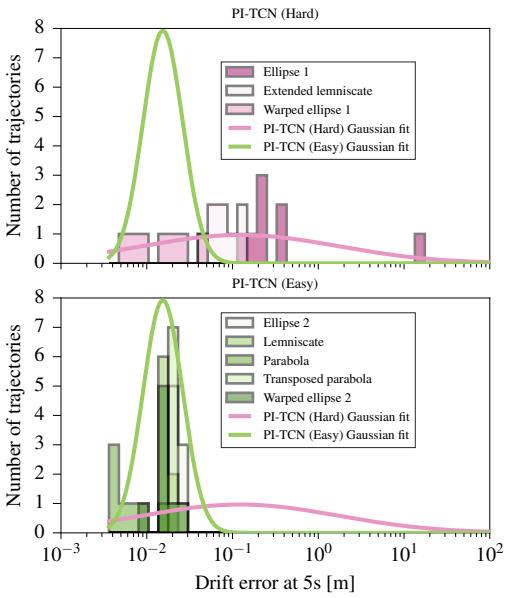


Figure 6: Drift error distribution of the TCN model across different PI-TCN sub-datasets. Each sample represents the drift error at a horizon of five seconds. We tested each sub-dataset's trajectories with eight seeds. The top and bottom axes show, respectively, the distributions of the easy and hard splits of PI-TCN. The solid line represents the Gaussian distribution fitted to the corresponding color group.

To test this hypothesis, we evaluated PI-TCN drift errors for each sub-dataset defined in the PI-TCN dataset. Fig. 6 shows the histogram of drift error with a 5-second horizon for all sub-datasets. We fix the horizon at five seconds for this analysis, matching the horizon used for the rest of the paper's results. The top view presents the drift error obtained when clustering the sub-datasets: Ellipse 1, Warped ellipse 1, and Extended lemniscate. The remaining datasets are presented on the bottom axis. The solid line represents the Gaussian curve fitted to the top-axis sub-datasets (pink) and the bottom-axis sub-datasets (green). The solid line shows that the two clustered sub-datasets have

different error orders. To eliminate this bimodality in the drift error, we split the Pi-TCN sub-datasets into two splits. The top axis in pink is the PI-TCN Hard split. The bottom axis in green is the PI-TCN Easy split. In conclusion, the PI-TCN dataset was split into two subgroups, resulting in a bimodal error distribution of the most precise baseline.

G Experimental Setup Details

PI-TCN easy/hard. A dataset from Saviolo et al. (2022), which contains 68 quadrotor trajectories including straight-line accelerations, circles, ellipses, lemniscates, and parabolic maneuvers (See Fig. 7 for trajectory examples). These trajectories reach speeds of up to $6\,\mathrm{m/s}$ and linear accelerations of up to $18\,\mathrm{m/s^2}$, totaling $58\,\mathrm{minutes}$ of flight data recorded at $100\,\mathrm{Hz}$. Pose is provided by a Vicon motion-capture system, while velocities and accelerations are obtained by fusing measurements from the on-board sensors with the Vicon data. We observed that AR-TCN errors on PI-TCN are bimodal. Three trajectories (warped ellipse, extended lemniscate, transposed parabola) stand far above the rest. So that they do not dominate aggregate metrics, we report them separately as PI-TCN *hard*, and the rest as PI-TCN *easy* (See Section F for split analysis).

Table 5: PI-TCN dataset (long-horizon release CSVs, 100 Hz).

CSV Field	Description	Source	Offline processing
`t`	Timestamp (s)	Vicon	—
`p_{x,y,z}` (world)	Position (m)	Vicon	—
`q_{w,x,y,z}` (world)	Orientation (quaternion)	Vicon	4th-order Butterworth low-pass, 5 Hz cutoff
`v_{x,y,z}` (world)	Linear velocity (m/s)	Vicon+Onboard UKF	—
`w_{x,y,z}` (body)	Angular velocity (rad/s)	Vicon+Onboard UKF	—
`vdot_{x,y,z}` (world)	Linear acceleration (m/s^2)	Computed offline	Derived from UKF-filtered linear velocity
`wdot_{x,y,z}` (body)	Angular acceleration (rad/s^2)	Computed offline	Derived from UKF-filtered angular velocity
`u_{0,1,2,3}`	Rotor speeds (rpm)	Onboard	4th-order Butterworth low-pass, 5 Hz cutoff
`f_{0,1,2,3}`	Rotor thrusts (N)	Computed offline	$f_i = k_f u_i^2$, $k_f = 4.379 \times 10^{-9}$

NeuroBEM A dataset from Bauersfeld et al. (2021), which contains 96 quadrotor flights, including lemniscates, ellipses, slanted circles, linear oscillations, race tracks, melon-shaped paths and random-point trajectories. These flights total $1.25\,\mathrm{hours}$ of data and reach speeds of up to $18\,\mathrm{m/s}$ and accelerations of up to $46.8\,\mathrm{m/s^2}$. Pose is recorded at $400\,\mathrm{Hz}$ by a Vicon motion-capture system, while inertial measurement unit (IMU) and motor-speed measurements are logged on board. Both streams are synchronized offline and fused by fitting cubic splines, which are then differentiated to recover linear velocity and angular acceleration. Compared to PI-TCN *easy*, NeuroBEM covers a far more aggressive flight envelope, is sampled at higher rates and is more heavily smoothed, as seen on Table 6.

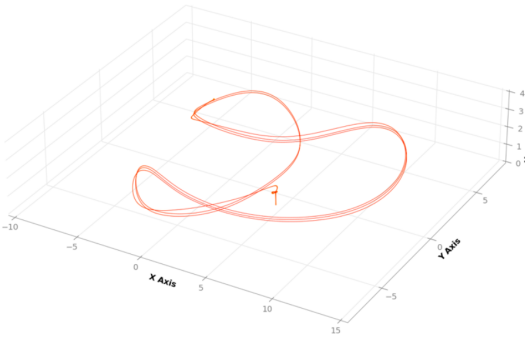


Figure 7: Uncrewed aerial vehicle (UAV) trajectory example.

Husky A dataset collected specifically for this work with a Clearpath Husky A200 following the Data-driven Robot Input Vector Exploration (DRIVE) protocol (Baril et al., 2024). The protocol randomly samples commands over the robot's input space, which covers the whole command space without bias and excites a wide range of accelerations. Each command is held for $6\,\mathrm{s}$ in open loop.

Table 6: NeuroBEM dataset (original processed data, 400 Hz).

CSV Field	Description	Source	Offline processing
`t`	Timestamp (s)	Vicon+Onboard	Time syncing (offset + clock skew), resampled to 400 Hz.
`pos {x,y,z}` (world)	Position (m)	Vicon	Fitted cubic spline, outlier rejection.
`quat {x,y,z,w}` (world)	Orientation (quaternion)	Vicon	Fitted cubic spline, outlier rejection.
`vel {x,y,z}` (body)	Linear velocity (m/s)	Vicon derived	1st derivative of position splines.
`ang vel {x,y,z}` (body)	Angular velocity (rad/s)	Onboard gyroscope	Fitted cubic splines.
`ang acc {x,y,z}` (body)	Angular acceleration (rad/s^2)	Onboard gyroscope derived	1st derivative of angular velocity splines.
`acc {x,y,z}` (body)	Linear acceleration (m/s^2)	Vicon derived	2nd derivative of position splines.
`mot {1,2,3,4}`	Rotor speeds (rad/s)	Onboard	4th-order Butterworth low-pass, cutoff $1/\tau_m \approx 30$ Hz.

Unlike in quadrotor datasets, deviations from the path are never corrected. The dataset totals $1.4\,\mathrm{h}$ of driving on grass, gravel, and snow, with speeds of up to $1\,\mathrm{m/s}$ and accelerations of up to $2.5\,\mathrm{m/s^2}$. The robot 3D pose is estimated at 10 Hz by lidar odometry based on iterative closest point (ICP). We release three versions of the data: `online` uses the odometry computed on board during collection, `offline` recomputes it afterwards to ensure a laser scan is never dropped, and `smoothed` further filters the offline pose before differentiating it into velocities to reduce noise. The details are in Table 7. The current work shows results on the `online` version on grass and gravel terrain.

Table 7: Husky DRIVE dataset fields for the `online`, `offline`, and `smoothed` release CSVs ($\sim$10 Hz).

		Offline processing		
CSV Field	**Source**	`online`	`offline`	`smoothed`
`timestamp_ns`	Computed offline	Online ICP odometry stamps	Offline ICP odometry stamps	Offline ICP odometry stamps
`pos_{x,y,z}` (world)	Lidar (ICP)†	–	–	CCMA smoothing
`quat_{x,y,z,w}` (world)	Lidar (ICP)†	–	–	Moving average (window 5) on unwrapped Euler angles
`l_vel_{x,y,z}` (world)	Lidar-derived	Finite difference of position	Finite difference of position	1st derivative of cubic spline fitted to smoothed position
`a_vel_{x,y,z}` (body)	Lidar-derived	Finite difference of orientation	Finite difference of orientation	Finite difference of smoothed orientation
`steering` (rad/s)	Onboard command	–	–	–
`speed` (m/s)	Onboard command	–	–	–

†The difference between online and offline ICP is that the online ICP can drop scans because of onboard processing when the offline ICP takes every scan

Lorenz A dataset is simulated from the Lorenz system (Lorenz, 1963), a three-dimensional model of atmospheric convection and the canonical example of deterministic chaos (See Fig. 8 for trajectory examples). The system is fully deterministic and its state stays bounded on the well-known butterfly-shaped attractor, yet it never repeats itself: the state circles one lobe of the butterfly, then jumps to the

other at irregular, hard-to-predict moments. What makes it special for our purpose is its sensitivity to initial conditions. Two trajectories starting arbitrarily close to each other drift apart exponentially fast, so any prediction error, however small, compounds until the predicted and true trajectories have nothing in common anymore. Unlike the robotic datasets, there is no controller or command input to pull the state back toward a reference or to excite it from outside; errors grow on their own instead of being damped. This places Lorenz at the far end of the sensitivity spectrum and makes it a stress test for compounded prediction.

Our setting is designed to make this stress test as clean as possible. Being synthetic, the ground truth is exact and the measurement noise is set by construction, so the compounding of errors observed at test time can only come from the models themselves. All trajectories start from the same initial coordinates and share the same parameters except a carefully chosen one in the sixth decimal: this is enough for the trajectories to decorrelate on the attractor after a short time, which is exactly the chaotic behavior we want to expose the models to. Since there is no control input, we feed the time step between two samples to the action feature dimension, and the models observe the time derivative of the state rather than the state itself.

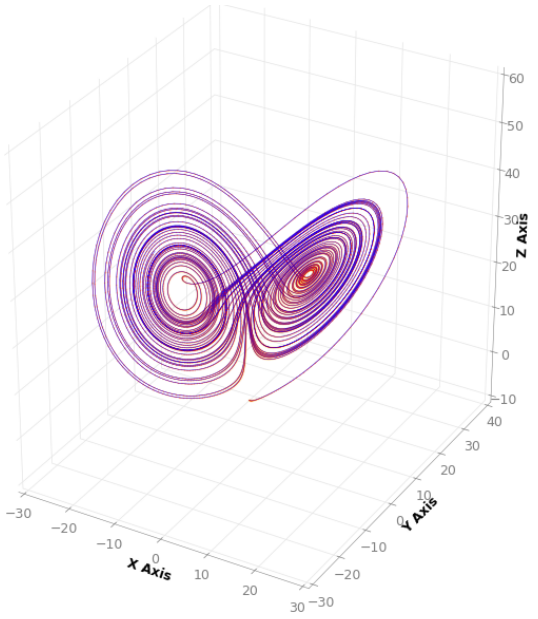


Figure 8: Lorenz attractor trajectory example.

Remark: In the results section, the Lorenz system was scaled by a factor of 2.5.

The simulator configuration used for all Lorenz experiments is consolidated in Table 8. Each source trajectory is generated by forward-Euler integration of the Lorenz equations over the time interval $[0, 60]$ discretized into $10\,000$ timesteps ($\Delta t = 0.006$), starting from the same initial coordinates $(x_0, y_0, z_0) = (0.0, 1.0, 1.05)$. Since the system has no control input, the dataset diversity comes from the dynamics parameter rather than from the command: all trajectories share $\sigma = 10.09$ and $\beta = 2.667$, and differ only by their Rayleigh number ρ. The observation fed to the models is the time derivative of the state, $(\dot{x}, \dot{y}, \dot{z})$, together with the delta time, and every observation is corrupted by zero-mean Gaussian measurement noise of standard deviation 10^{-5} over the whole trajectory, both at training and at deployment time. Five source trajectories of $10\,000$ timesteps each ($50\,000$ timesteps in total) are used for training and validation, and three distinct in-distribution trajectories of $10\,000$ timesteps each ($30\,000$ timesteps in total) are reserved for test-time deployment. The test trajectories share the initial coordinates, σ, β, the noise level and the time discretization of the training set and differ only by their ρ value, one of which ($\rho = 28.000011$) coincides with a training trajectory, the other two falling in between training values. No out-of-distribution trajectory is used in the current work.

H Inference rate analysis

Table 9 reports the median control-loop inference rate of every model on the embedded computer, on CPU and on GPU, with its inter-quartile spread; PI-TCN and Neurobem are regrouped under the UAV column because they share the same feature vector and horizon, thus the same rate. All rates were collected on an NVIDIA Jetson AGX Orin running a kernel compiled for real-time, with the Jetson clocks enabled and the CPU frequency governor set to `performance` rather than `schedutil`, by feeding each model synthetic inputs of the same type and dimension as its target dataset.

On the small observation/action regime of our datasets (a history of 20 steps, a horizon of 10, a handful of observation and action features, the control-loop step of a deployed world model is dominated by bookkeeping rather than by arithmetic: feature indexing and the history-buffer update

Table 8: Lorenz simulator configuration (`lorenz_v6_5XinitC_low_global_noise`).

Setting	Value
Dynamics	Lorenz system, forward-Euler integration
Initial coordinates (x_0, y_0, z_0)	$(0.0,\ 1.0,\ 1.05)$ for every trajectory
Shared parameters	$\sigma = 10.09$, $\beta = 2.667$
Time interval	$[0, 60]$, 10 000 timesteps,
Observation	State time derivative $(\dot{x}, \dot{y}, \dot{z})$ and delta time
Measurement noise	Zero-mean Gaussian, $\sigma_{\text{noise}} = 10^{-5}$, applied to every timestep, identical at training and deployment
Explorable region	Full trajectory ($[0, 10\,000]$)
Train/validation trajectories	5 trajectories $\times$ 10 000 timesteps (50 000 total)
Non-shared parameter ρ	28.000001, 28.0000025, 28.000004, 28.000007, 28.000011
Test-time deployment trajectories	3 in-distribution trajectories $\times$ 10 000 timesteps (30 000 total)
Non-shared parameter ρ	28.000005, 28.000009, 28.000011
Out-of-distribution trajectories	None

Table 9: Inference-rate comparison across datasets. **Bold** highlights the key take-away: on both robotic datasets, MTM-Pro on CPU alone outruns the previous state of the art in accuracy, AR TCN, on GPU.

Type	Method	UAV dataset		Husky		Lorenz	
		Inf. CPU [Hz]	**Inf. GPU** [Hz]	**Inf. CPU** [Hz]	**Inf. GPU** [Hz]	**Inf. CPU** [Hz]	**Inf. GPU** [Hz]
Auto-Regressive	MLP	$294^{+1.7}_{-1.9}$	$224^{+0.81}_{-0.91}$	$321^{+1.9}_{-2.0}$	$276^{+1.2}_{-1.2}$	$416^{+3.1}_{-3.0}$	$275^{+1.2}_{-1.1}$
	GRU	$268^{+1.7}_{-1.5}$	$222^{+0.86}_{-0.88}$	$298^{+2.4}_{-2.2}$	$274^{+1.2}_{-1.2}$	$376^{+2.4}_{-2.7}$	$274^{+1.1}_{-1.1}$
	LSTM	$256^{+1.2}_{-1.2}$	$222^{+1.1}_{-0.76}$	$280^{+1.6}_{-1.7}$	$274^{+1.1}_{-1.3}$	$368^{+2.7}_{-2.8}$	$269^{+1.1}_{-1.2}$
	MLP-Prob	$242^{+1.5}_{-1.5}$	$173^{+0.63}_{-0.60}$	$270^{+1.5}_{-1.5}$	$212^{+0.76}_{-0.81}$	$325^{+2.6}_{-2.5}$	$201^{+0.73}_{-0.69}$
	TCN	$48.2^{+0.48}_{-3.1}$	$\mathbf{160^{+0.59}_{-0.50}}$	$49.6^{+0.89}_{-3.7}$	$\mathbf{184^{+0.68}_{-0.71}}$	$96.8^{+14}_{-6.3}$	$206^{+0.83}_{-0.85}$
Multi-step	E2E-TCN	$8.86^{+2.0}_{-1.8}$	$87.3^{+0.22}_{-0.23}$	$29.2^{+3.8}_{-1.1}$	$117^{+0.36}_{-0.41}$	$8.41^{+1.7}_{-1.5}$	$83.8^{+0.22}_{-0.24}$
	M3	$27.8^{+2.0}_{-4.5}$	$127^{+0.36}_{-0.37}$	$106^{+0.42}_{-0.71}$	$154^{+0.62}_{-0.61}$	$25.0^{+4.0}_{-4.2}$	$158^{+0.45}_{-0.50}$
Ours	MTM-Pro	$\mathbf{182^{+0.96}_{-1.1}}$	$126^{+0.37}_{-0.38}$	$\mathbf{208^{+0.91}_{-1.1}}$	$145^{+0.48}_{-0.43}$	$185^{+0.98}_{-1.4}$	$140^{+0.51}_{-0.50}$

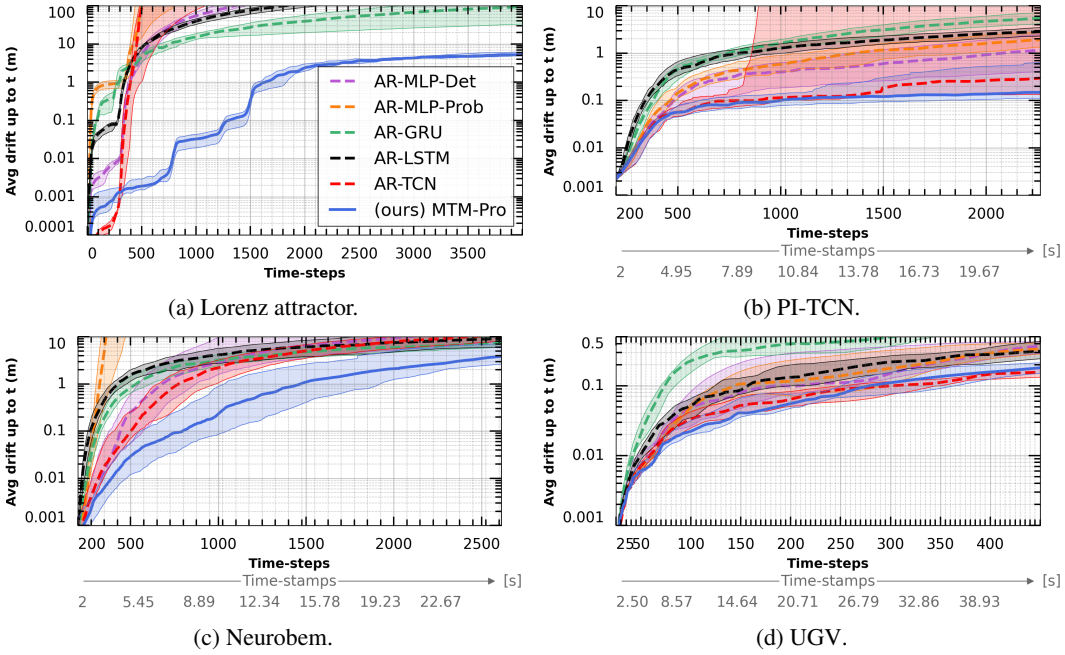
Abbreviations: Inf. CPU = Inference rate on CPU; Inf. GPU = Inference rate on GPU. Each cell is the median control-loop rate with its $[Q1, Q3]$ spread as sub/superscript. **Bold** values are the ones compared in the key take-away: the CPU rate of MTM-Pro against the GPU rate of AR TCN on the two robotic datasets (UAV and Husky), the former being higher in both cases; underlined values indicate the best result within their own Type group (Auto-Regressive / Multi-step / Ours).

cost as much as the forward pass, so the loop is CPU-bound for every non-convolutional model, at least on the dynamics-model side.

The lightweight architectures (Multi-Layer Perceptron (MLP), Gated Recurrent Units (GRU)) have an edge in this regime: they reach their full rate without GPU acceleration, hence on CPU-only platforms, whereas the convolutional models need the GPU to reach a useful rate. The key take-away (in bold in the table) is that on every robotic dataset, MTM-Pro running on the CPU alone is faster than AR TCN, the previous state of the art in accuracy, running on the GPU: 182 Hz against 160 Hz on the UAVs and 208 Hz against 184 Hz on Husky. MTM-Pro keeps the CPU-deployable profile of the lightweight models (Section 4.4), which leaves the GPU to the controller and the state estimator; in a model-based reinforcement-learning setting it also frees the accelerator for the training loop of the world model itself, one CPU core serving the model to the controller while the GPU is dedicated to learning. In short, MTM-Pro needs no GPU to outrun the previous state of the art.

I ADDITIONAL EXPERIMENTAL RESULTS

Per-dataset drift rate. Figure 9 reports the compounded-prediction drift rate of every method on each dataset, the curves behind the 5 s snapshots of Table 1 and Figure 2.



(a) Lorenz attractor.

(b) PI-TCN.

(c) Neurobem.

(d) UGV.

Figure 9: Compounded-prediction drift rate per dataset.