Learning Kalman Filters as Linear RNNs

Sultan Daniels, Gautam Goel, Wentinn Liao, Gireeja Ranade, Saagar Sanghavi, Anant Sahai

University of California, Berkeley

Abstract: Deep learning has nurtured a shared ecosystem of tools and practices facilitating the learning of various models from data. Although the established communities interested in system-identification and adaptive control/filtering have tended to use different tools for historical reasons, recently, deep learning tools are also being used for system identification. This paper shows how to leverage the PyTorch ecosystem for learning simple linear filters from data using a "next token prediction" self-supervised loss objective popularized by the training of GPT-style models in the field of natural-language processing. We show how empirically, these learned linear filters converge towards the analytic steady-state Kalman Filter as we increase the amount of training data. We also show that time-series oriented Transformer-based models also show the same qualitative scaling of prediction performance with training data, although the actual prediction performance is significantly worse with the same amount of training data.

Keywords: Kalman Filters; PyTorch; Sample Complexity; Self-supervision; Next-output prediction; Transformers

1. INTRODUCTION

Kalman filters are a fundamental and longstanding building block of more complicated systems. This paper explores how to leverage standard deep learning (DL) tools (Paszke et al., 2019) together with a version of the "next-token-prediction" semi-supervised learning objective recently popularized by GPT models (Radford et al., 2018) to learn filters from data. We show empirical results for two example systems that suggest convergence to the ideal analytic Kalman filter (also used as a next-observation predictor (Kailath et al., 2000) with excess mean-squared-error decaying somewhere between $O(1/N)$ and $O(1/\sqrt{N})$ with the length N of the observation trace.

Despite the well-established nature of Kalman filters and LQR control, the area is undergoing a significant revival of interest [1] in light of data-efficient adaptation and learning. On the theoretical side, the focus of such research has been on giving asymptotic and finite-data convergence guarantees for learning the Kalman filter. Here the focus is often on specific achievable rates of algorithms (e.g. certainty equivalence). In this paper, we take an experimental approach — what is the best we can do if we try use a standard Recurrent Neural Network Framework and gradient descent to learn the filter. Given the tremendous progress in our ability to use data-driven methods, and the vast improvements in the tool ecosystem, it is important to understand the possibilities and limitations of modern machine learning tools for both system identification and sequence prediction.

A conceptually central problem is understanding how to make the best use of the limited data that we have. Deistler (2001) provides an overview of the standard approach for system identification, which has largely had a focus on asymptotic guarantees in the limit of large sample size. Sayed (2001) proposes a new framework for a robust Kalman filter for an unknown system in the presence of parametric uncertainty on the system parameters. However, given the reality of finite data, more recent results have focused on approaches that try to more accurately capture the finite-data regime, for instance Abbasi-Yadkori and Szepesvári (2011); Tsiamis et al. (2020); Tsiamis and Pappas (2022).

Hardt et al. (2018) showed that we can use gradient descent to learn a dynamical system with polynomial convergence rates and sample complexity. Dean et al. (2020) and related papers studied the sample complexity of the LQR problem, while Abbasi-Yadkori and Szepesvári (2011) looked at LQR from a regret perspective. While LQR and Kalman Filtering are dual problems when the system dynamics are perfectly known, this is not true in the case we consider of unknown dynamics. Partial observability can make the Kalman Filtering problem more challenging in the case of unknown dynamics.

Tsiamis et al. (2020) used a two-step identification process to learn a Kalman filter for an unknown system. In this, they first focus on learning the system parameters and then compute a certainty-equivalent style Kalman filter for the learned system. Additionally, they provide a more robust approach using system-level synthesis as well. They find that under certain conditions, the mean-squared error decays (with high probability) as $\tilde{O}(1/\sqrt{N})$, where N is number of training samples. However, it is known that such a two-step estimation process for the Kalman Filter can be subptiomal (El Ghaoui and Calafiore, 2001), and we see this in our experiments, where in some cases the convergence is much faster empirically. Hazan et al. (2018) consider a similar problem but with adversarial noise, and show they can achieve logarithmic regret, independent of the spectral radius/eigenvalues of the matrix A. Tsiamis and Pappas (2022) show that they can use linear regression on the observations to achieve logarithmic regret with respect to the Kalman filter. Their approach for this is similar to that of Kozdoba et al.

[1] The Kalman filter is also used extensively in more traditional deep-learning applications such as video analysis to be able to track objects Fraccaro et al. (2017). Consequently, interest in advances here comes from both directions.

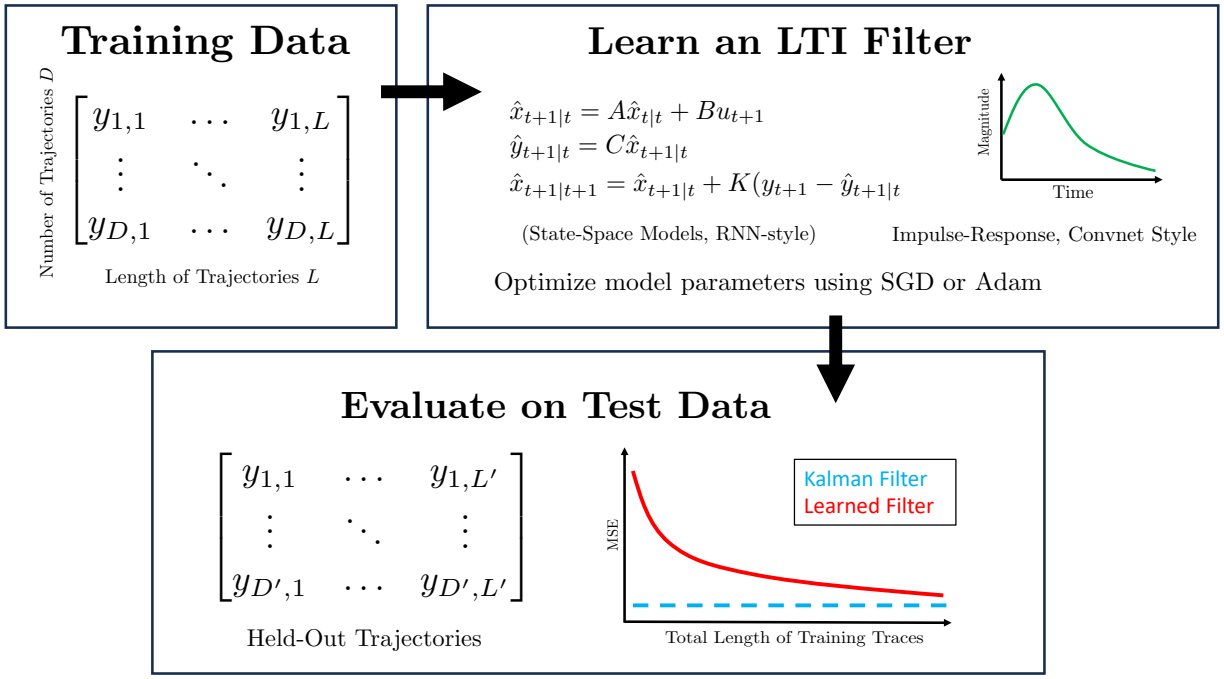

Fig. 1. Overview of our training and evaluation pipeline. For a fixed system, we sample N observation trajectories of length M each. A state-space model for a filter to be learned is represented by an RNN. To evaluate our learned models, we measure their performance relative to the ideal Kalman filter on held-out trajectories from the same system.

(2019), who model the Kalman filter output as the regression of a past few observations, and show that under certain conditions, this can approximate the Kalman filter very well.

On the practical side, combining deep-learning and system identification has been gathering interest recently as well, see Ljung et al. (2020); Pillonetto et al. (2023) for comprehensive reviews on this subject. There have been substantial efforts to create tools that can be broadly used for system identification of dynamical systems. For example, a series of works Forgione and Piga (2021a,b); Forgione et al. (2023); Piga et al. (2021); Mavkov et al. (2020); Beintema et al. (2021) have led to the powerful Github repository `pytorch-ident`. Andersson et al. (2019) show how convolutional networks can be used to do system identification in addition to other approaches.

Here, we adapt established DL techniques to learning a linear filter that predicts a linear system's output given past outputs and controls. For comparison, we also show how off-the-shelf Transformer-based tools perform for the same simple systems:

Example Linear Models Used for Experiments

$$\mathbf{x}_{i+1} = A\mathbf{x}_i + B\mathbf{u}_i + \mathbf{w}_i \tag{1}$$
$$\mathbf{y}_i = C\mathbf{x}_i + \mathbf{v}_i \tag{2}$$

where $\mathbf{x}_i \in \mathbb{R}^n$, $\mathbf{w}_i \in \mathbb{R}^m$, $\mathbf{u}_i \in \mathbb{R}^k$, $\mathbf{v}_i \in \mathbb{R}^p$, and $\mathbf{y}_i \in \mathbb{R}^p$. The matrices A, B, and C are sized accordingly. Also, $\mathbf{w}_i \sim \mathcal{N}(0, W)$, $\mathbf{v}_i \sim \mathcal{N}(0, V)$. Additionally, $\mathbb{E}\left[\mathbf{w}_i \mathbf{v}_i^T\right] = 0$, and for all $i \neq j$, $\mathbb{E}\left[\mathbf{w}_i \mathbf{w}_j^T\right] = \mathbb{E}[\mathbf{v}_i \mathbf{v}_j] = 0$. Finally, $\mathbf{w}_i$ and $\mathbf{v}_i$ are uncorrelated with all past or present states ($\mathbf{x}_j$ where $j \leqslant i$).
In both of our examples, we have $B = \mathbf{0}, W = 0.10I, V = 0.01$.

Example 2-Dimensional System:

$$A \approx \begin{bmatrix} 0.96 & 0.16 \\ -0.03 & 0.80 \end{bmatrix}, \; C \approx [0.83 \;\; 0.94].$$

Example 6-Dimensional System:

$$A \approx \begin{bmatrix} 0.67 & 0.03 & -0.13 & -0.23 & -0.03 & 0.32 \\ 0.03 & 0.54 & -0.01 & -0.04 & 0.30 & -0.26 \\ -0.01 & -0.27 & 0.67 & -0.038 & 0.12 & 0.05 \\ 0.08 & -0.05 & 0.06 & 0.31 & 0.07 & 0.13 \\ 0.10 & -0.01 & -0.17 & 0.00 & 0.80 & -0.11 \\ 0.08 & -0.02 & -0.08 & -0.07 & 0.19 & 0.49 \end{bmatrix},$$
$$C \approx [0.42 \;\; 0.52 \;\; 0.43 \;\; 0.00 \;\; 0.94 \;\; 0.33].$$

2. RNN-STYLE FILTER LEARNING

Recurrent Neural Nets (RNNs) have been around for almost four decades Rumelhart et al. (1986) and were directly inspired by the classical state-space perspective on IIR filters. The steady-state (time-invariant) Kalman Filter is therefore exactly realizable as a linear (i.e. one without any nonlinear activation function) RNN. Because RNNs are a standard DL building block, they are well supported by standard DL frameworks like PyTorch Paszke et al. (2019) that provide programming support for automatic differentiation, the use of various optimizers, and easier leveraging of computational resources like GPUs. In this section, we describe exactly how we implement the learning of filters from data, as well as how we both generated this data from nominal linear systems and evaluated the performance of these learned filters.

2.1 Basic Implementation

One key to leveraging DL frameworks like PyTorch is specifying the underlying parameter space of learnable matrices that will be optimized during training. Our implementation of the RNN possesses an internal state $\hat{x}$, and four learnable matrices,

$$A_\theta \in \mathbb{R}^{d_{state} \times d_{state}} \qquad B_\theta \in \mathbb{R}^{d_{state} \times d_{control}} \tag{3}$$
$$C_\theta \in \mathbb{R}^{d_{observation} \times d_{state}} \qquad K_\theta \in \mathbb{R}^{d_{state} \times d_{observation}} \tag{4}$$

In forward propagation, it takes a control $u \in \mathbb{R}^{d_{control}}$, and an observation $y \in \mathbb{R}^{d_{observation}}$, updating its internal state and outputting an observation prediction $\hat{y}$. Formally, to move from timestep t to $t+1$, its internal state starts denoted $\hat{x}_t$ and the code performs the following operations [2]:

$$\hat{x}_{t+1|t} = A_\theta \hat{x}_t + B_\theta u_t \tag{5}$$

$$\hat{y}_{t+1|t} = C_\theta \hat{x}_{t+1|t} \qquad \text{(output)} \tag{6}$$

$$\hat{x}_{t+1} = \hat{x}_{t+1|t} + K_\theta (y_{t+1} - \hat{y}_{t+1|t}) \tag{7}$$

2.2 Training

Generating the training data: We randomly draw an initial state $x_0 \sim \mathcal{N}(0, \frac{1}{d_{state}} I) \in \mathbb{R}^{d_{state}}$ so that the norm-squared of the initial state has expected value 1. A sequence of controls $(u_t)_{t=0}^{L-1}$ are similarly generated i.i.d. with $u_t \sim \mathcal{N}(0, \frac{1}{d_{control}} I)$ for $0 \leqslant t < L$ where L denotes the length of an individual training trace. These inputs along with the initial state and i.i.d. observation noises are used to drive an underlying linear state-space model to produce the ground-truth observations $(y_t)_{t=1}^{L}$. Doing so in batch produces a set of D training *traces* of length L, denoted

$$\mathscr{D} = \left\{ (u_{d,t-1}, y_{d,t}) \;\middle|\; 1 \leqslant d \leqslant D, 1 \leqslant t \leqslant L \right\},$$

These can be implemented in PyTorch as two tensors $u \in \mathbb{R}^{D \times L \times d_{control}}$ and $y \in \mathbb{R}^{D \times L \times d_{observation}}$.

Although we know a ground-truth linear state-space model while generating the training data, this is not given to the learning process below.

Training the RNN-style model Within the modern empirical machine learning paradigm, there are four key questions that have to be answered to understand how we train:

(1) What are we trying to optimize? What is the loss function? For this, we follow the lead of the GPT-1 Radford et al. (2018) model and use what is called the self-supervised loss of next-symbol prediction, where distance is measured using squared-loss. Specifically, for any given trace of length L, the training loss is

$$\ell(\hat{y}, y) = \frac{1}{L} \sum_{t=1}^{L} \|\hat{y}_t - y_t\|_2^2$$

(2) What optimizer are we using? For this, although we did experiments using stochastic gradient descent with momentum as well, we use the AdamW optimizer Kingma and Ba (2014); Loshchilov and Hutter (2018) with a learning rate schedule to present results here. The exact optimizer hyperparameters and related discussion are in Section 2.4.

(3) How do we initialize our parameters? Our main results use what we call the identity initialization:

$$A_\theta = I_{d_{state}},\ B_\theta = 0_{d_{state}, d_{control}}$$
$$C_\theta = 0_{d_{observation}, d_{state}},\ K_\theta = 0_{d_{state}, d_{observation}}.$$

(4) How do we draw batches for training? The remainder of this subsection answers that question.

Because the case where $D = 1$ (learning from a single long trace) is practically important [3], we uniformly sample contiguous subsequences of the training traces, where the length of the subsequence is denoted by L_{SS}. Formally, a batch is given by

$$\mathscr{D}_{batch} = \left\{ (u_{d_i, t-1}, y_{d_i, t}) \;\middle|\; 1 \leqslant i \leqslant B, t_i \leqslant t < t_i + L_{SS} \right\},$$

where d_i are i.i.d. integers uniformly sampled from $1, \ldots, D$ and t_i are i.i.d. integers uniformly sampled from $1, \ldots, L - L_{SS} + 1$ which are also independent of d_i. In implementation, this batch materializes as two tensors $u_{batch} \in \mathbb{R}^{B \times L_{SS} \times d_{control}}$ and $y_{batch} \in \mathbb{R}^{B \times L_{SS} \times d_{observation}}$.

Because the RNN must be initialized with an internal state, during each training iteration we randomly reinitialize the internal state as $\hat{x}_0 \sim \mathcal{N}(0, \frac{1}{d_{state}} I)$, while during validation we always initialize it to $\hat{x}_0 = 0_{d_{state}}$.

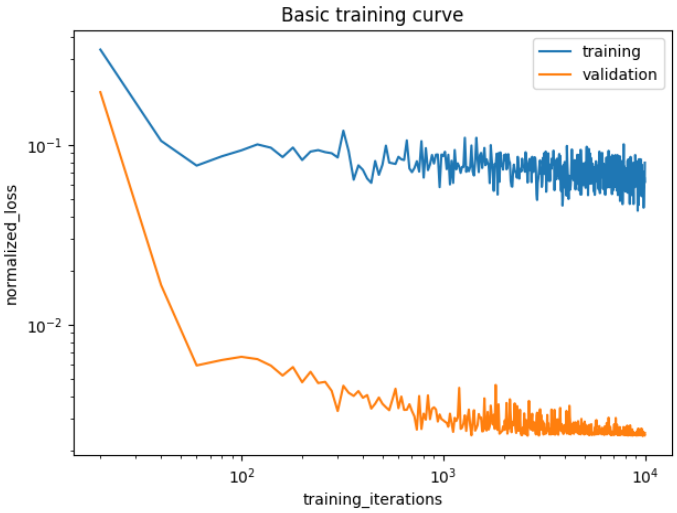

Fig. 2. Log-log training curve of an RNN model on the 2-dimensional system with a total wallclock time of 138.158s. The y-axis is the normalized loss (both training and validation), while the x-axis is the number of training iterations.

2.3 Analysis

To analyze the performance of the learned filters, we evaluate these filters on held-out traces drawn from the same underlying linear model, but with different realizations for the initial conditions, driving noise, and observation noise. We can compare the avg-squared-error (MSE) loss with two key metrics.

The first is the theoretical irreducible loss that is achieved in expectation by the analytical Kalman filter. If we denote the analytical Kalman filter as a function KF and a trace as τ, then this metric is given by

$$\frac{1}{L} \mathbb{E}_\tau \left[\|KF(\tau) - \tau\|_2^2 \right].$$

This analytic quantity (when $L \to \infty$) provides a ballpark of what our learned models "should be converging to." For the specific system of state-dimension 2, this value is 0.176441 and for that of state-dimension 6, this value is 0.221683.

The second is the empirical irreducible loss. While the MSE of the analytical Kalman filter will achieve the aforementioned irreducible loss in expectation, on any given random trace, the actual performance of the Kalman filter is a random variable. Because variance in the traces themselves causes variance in the MSE of the learned Kalman filter which may in turn perturb

[2] Note that it is important to make the observation prediction before it receives the true observation for the next time step.

[3] Memory limitations on GPUs also constrain the length of traces that we can load for a single optimization step.

our analysis, we reduce this variance stemming from trace randomness by subtracting the MSE of the analytical Kalman Filter. Denote the learned filter as F_θ. We can now evaluate it on a trace τ by calculating

$$\frac{1}{L}(\|F_\theta(\tau)-\tau\|_2^2-\|KF(\tau)-\tau\|_2^2),$$

which we deem the *normalized loss*. To get a good estimate of this number, we evaluate it on a large held-out dataset of 1000 traces, each of length 1000.

To see the impact of the randomness in the training data as well as the stochasticity of the optimization process itself, for each experiment we train 32 models in parallel and take the median normalized validation loss which provides a more stable metric for analysis that also allows us to visualize the impact of randomness. In figures 3 and 4, the darker and lighter regions show the quartile ranges with respect to training and validation respectively. We can see by the best fit line that the normalized validation loss exhibits an approximate $\frac{1}{n}$ decay with the size of training data, although we observe that the slope is marginally deeper. The straight-line negative slope on a log-log plot strongly suggests that given enough training data, the filter we learn will converge [4] to the analytic Kalman filter.

To understand the underlying optimization landscape a bit better, we can initialize our model to *start* at the analytical Kalman filter itself by setting A, B, and C to be those of the system, and K to be the time-invariant innovation matrix of the analytical Kalman filter, i.e.

$$A_\theta = A,\ B_\theta = B,\ C_\theta = C,\ K_\theta = K_\infty.$$

These curves are also plotted in figures 3 and 4 where they show the same behavior (which tells us that the main source of generalization error is the impact of the randomness in the training data itself), but surprisingly do a bit *worse* [5] than identity initialization!

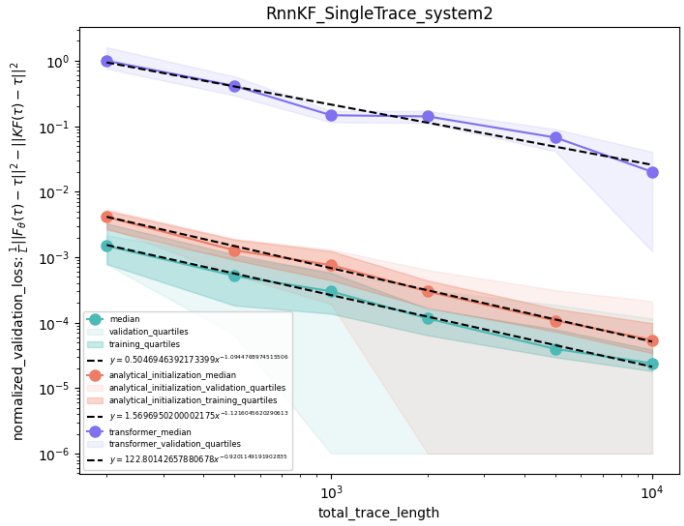

Fig. 3. Normalized validation loss as a function of training sequence length for an example 2-dimensional system. Notice the Transformer curve following along above.

[4] This convergence has to be in input-output behavior since many different parameters can give rise to the same input-output behavior.

[5] At the moment, we have no explanation for this behavior. It merely hints that the full story here is more subtle than we all might have suspected.

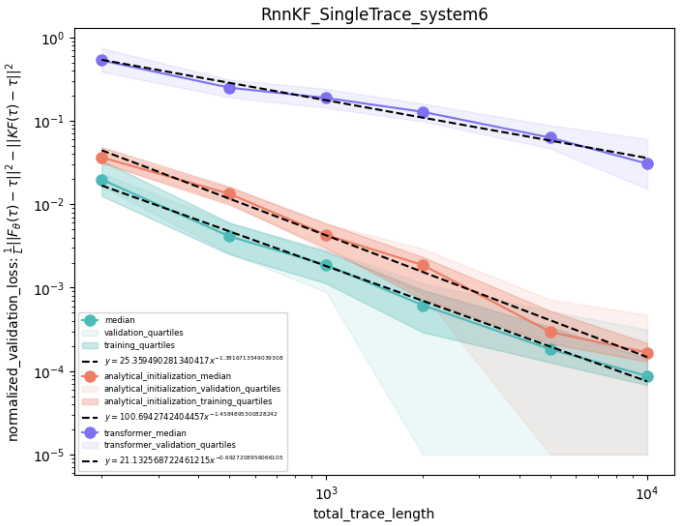

Fig. 4. Normalized validation loss as a function of training sequence length for an example 6-dimensional system. Notice the Transformer curve following along above.

2.4 *Hyperparameter choices*

In optimizing the performance of RNNs as learned Kalman filters, we find ***subsequence length*** and ***learning rate decay*** to be the most sensitive and hence important to get right. Shown below is a table of hyperparameters used to learn the two aforementioned systems.

Hyperparameter	2-dimensional	6-dimensional
Subsequence length	**16**	**48**
Learning rate decay (α)	**0.96**	**0.96**
Optimizer	AdamW	AdamW
Maximum learning rate (ρ_{max})	$2\cdot10^{-2}$	$2\cdot10^{-2}$
Minimum learning rate (ρ_{min})	$1\cdot10^{-6}$	$1\cdot10^{-6}$
Momentum	0.9	0.9
Weight decay	0.1	0.1
Batch size	128	128
Training iterations per epoch	100	100
Number of epochs	300	300

The (modified exponential) learning rate schedule is given by

$$\rho_n = \begin{cases} \rho_{max} & \text{if } 1 \leqslant n \leqslant 20 \\ \max(\rho_{max}\alpha^{n-20}, \rho_{min}) & \text{if } 20 < n \end{cases}$$

where ρ_n is the learning rate during epoch n.

Figures 5 and 6 illustrate the impact of subsequence length. We can see that for a long training sequences of length 5000-10000, a subsequence length of 16 is best for the example system with dimension 2 and a length of around 32 is best for the example with dimension 6. In general, a subsequence length that is sufficiently long reduces the stochasticity in batches and increases the likelihood of getting stuck in local minima, which may explain why the loss may eventually increase with subsequence length despite longer subsequences providing better gradients.

Similarly, Figure 7 shows that for training sequences of length at least 1000, a learning rate decay of 0.96 performs significantly better [6] for the system of dimension 2.

[6] The counterpart plots for the 6-dim system show far less of an impact.

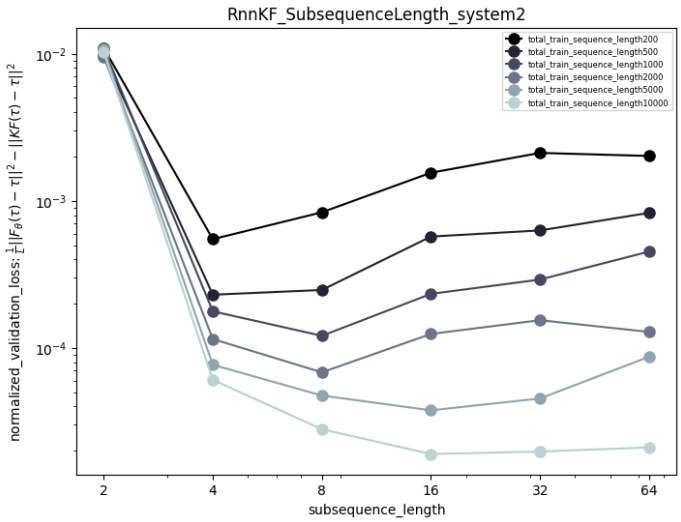

Fig. 5. Normalized validation loss over different train sequence lengths, and different batch subsequence lengths for system of state dimension 2.

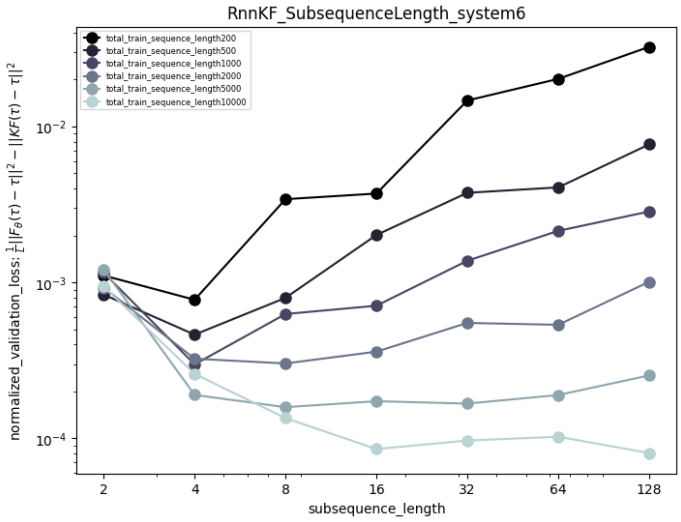

Fig. 6. Normalized valdation loss over different train sequence lengths, and different batch subsequence lengths for system of state dimension 6.

2.5 *Vectorization*

In order to run hundreds of experiments efficiently on GPUs, we had to leverage PyTorch's support [7] for vectorization. Beyond existing packages, we can further vectorize the process to exploit GPU parallelism by leveraging the underlying linearity and computing the lower triangular matrix of impulse responses for the initial state, controls, and observations.

Below we denote $(I-KC)A = M$. The proposed vectorization is given by the following:

[7] For running multiple models simultaneously, we leveraged the built-in ensembling functionality provided by PyTorch through functions such as `torch.func.vmap`, `torch.func.stack_module_state`, `torch.func.functional_call` and the `TensorDict` package.

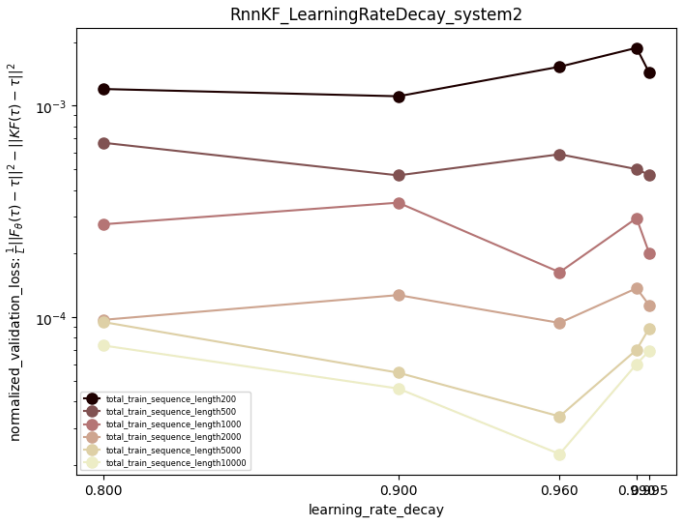

Fig. 7. Normalized validation loss over learning rate decays and different subsequence lengths for the 2d system.

$$M^{(L)} = \begin{bmatrix} I & \cdots & 0 & 0 \\ M & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ M^{L-1} & \cdots & M & I \end{bmatrix}$$

$$\begin{bmatrix} \hat{x}_1 \\ \hat{x}_2 \\ \vdots \\ \hat{x}_L \end{bmatrix} = \begin{bmatrix} M \\ M^2 \\ \vdots \\ M^L \end{bmatrix} \hat{x}_0 + M^{(L)} \left((I-KC)B \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{L-1} \end{bmatrix} + K \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_L \end{bmatrix} \right)$$

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_L \end{bmatrix} = C \left(A \begin{bmatrix} x_0 \\ \hat{x}_1 \\ \vdots \\ \hat{x}_{L-1} \end{bmatrix} + B \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{L-1} \end{bmatrix} \right)$$

Ideally with infinite vectorization, this would be computed in near-constant time. We can interpret this process as a function f that takes $u \in \mathbb{R}^{L \times d_{control}}$ and $y \in \mathbb{R}^{L \times d_{observation}}$ and outputs the impulse response of state $\hat{x}$ and observation estimation $\hat{y}$ with respect to the initial state x_0, i.e.

$$(\hat{x}[0:L], \hat{y}[0:L]) = f(u[0:L], y[0:L])(x_0)$$

where $f(u,y)$ can be constructed in $\mathcal{O}(1)$. However, the matrix $M^{(L)}$ takes up $\mathcal{O}(L^2)$ space, and with $L \geqslant 1024$ no longer fits in our GPU RAM.

We therefore compromise between sequential and vectorized propagation. Rather than constructing a single function $f(u_{0 \leqslant t \leqslant L-1}, y_{1 \leqslant t \leqslant L})$ that allows us to directly compute the end result, we instead construct a string of $\sqrt{L}$ functions given by

$$\left\{ f(u[t'\sqrt{L} : (t'+1)\sqrt{L}], y[t'\sqrt{L} : (t'+1)\sqrt{L}]) \right\}_{0 \leqslant t' < \sqrt{L}}$$

each of which can be computed in $\mathcal{O}(1)$ time, and allows the complete propagation to be computed in $\mathcal{O}(\sqrt{L})$ assuming ideal vectorization. As seen in figure 8, using vectorization to compute the impulse responses produces an approximate 26.9x speedup in training for long trace lengths. Based on the time comparison, we employ naive propagation if $L \leqslant 16$, single-function propagation if $16 < L \leqslant 512$, and multiple-function propagation if $512 < L$.

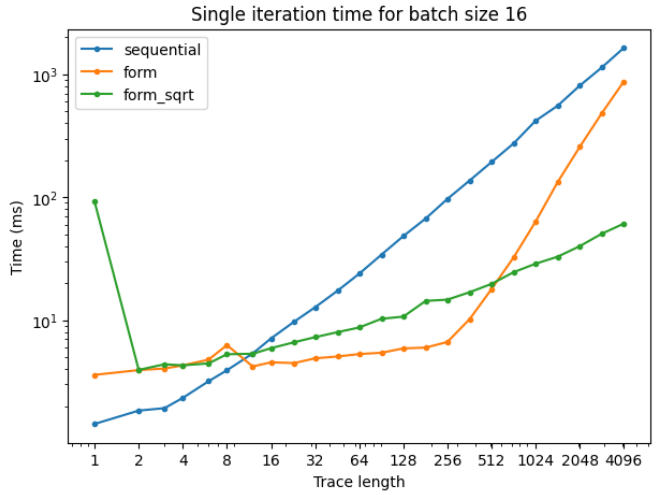

Fig. 8. The average time for a single iteration for all three methods of propagation.

3. TIME SERIES TRANSFORMER

The Hugging Face Transformers library contains a probabilistic time series transformer model (Rogge and Rasul, 2022). It is geared toward practitioners who want to make long horizon forecasts of the future values of the time series the model was trained on. By contrast, we only want to predict the immediate future. However, it is possible to adapt it for this problem, and we can see in figures 3 and 4 that the asymptotic scaling looks similar, even as the amount of data required is much larger.

REFERENCES

Abbasi-Yadkori, Y. and Szepesvári, C. (2011). Regret bounds for the adaptive control of linear quadratic systems. In *Proceedings of the 24th Annual Conference on Learning Theory*, 1–26. JMLR Workshop and Conference Proceedings.

Andersson, C., Ribeiro, A.H., Tiels, K., Wahlström, N., and Schön, T.B. (2019). Deep convolutional networks in system identification. In *2019 IEEE 58th conference on decision and control (CDC)*, 3670–3676. IEEE.

Beintema, G., Toth, R., and Schoukens, M. (2021). Nonlinear state-space identification using deep encoder networks. In *Learning for dynamics and control*, 241–250. PMLR.

Dean, S., Mania, H., Matni, N., Recht, B., and Tu, S. (2020). On the sample complexity of the linear quadratic regulator. *Foundations of Computational Mathematics*, 20(4), 633–679.

Deistler, M. (2001). System identification-general aspects and structure. In *Model identification and adaptive control: From windsurfing to telecommunications*, 3–26. Springer.

El Ghaoui, L. and Calafiore, G. (2001). Robust filtering for discrete-time systems with bounded noise and parametric uncertainty. *IEEE Transactions on Automatic Control*, 46(7), 1084–1089.

Forgione, M., Muni, A., Piga, D., and Gallieri, M. (2023). On the adaptation of recurrent neural networks for system identification. *Automatica*, 155, 111092.

Forgione, M. and Piga, D. (2021a). Continuous-time system identification with neural networks: Model structures and fitting criteria. *European Journal of Control*, 59, 69–81.

Forgione, M. and Piga, D. (2021b). dynonet: A neural network architecture for learning dynamical systems. *International Journal of Adaptive Control and Signal Processing*, 35(4), 612–626.

Fraccaro, M., Kamronn, S., Paquet, U., and Winther, O. (2017). A disentangled recognition and nonlinear dynamics model for unsupervised learning. *Advances in neural information processing systems*, 30.

Hardt, M., Ma, T., and Recht, B. (2018). Gradient descent learns linear dynamical systems. *The Journal of Machine Learning Research*, 19(1), 1025–1068.

Hazan, E., Lee, H., Singh, K., Zhang, C., and Zhang, Y. (2018). Spectral filtering for general linear dynamical systems. *Advances in Neural Information Processing Systems*, 31.

Kailath, T., Sayed, A.H., and Hassibi, B. (2000). *Linear Estimation*. Prentice Hall.

Kingma, D.P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kozdoba, M., Marecek, J., Tchrakian, T., and Mannor, S. (2019). On-line learning of linear dynamical systems: Exponential forgetting in kalman filters. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 4098–4105.

Ljung, L., Andersson, C., Tiels, K., and Schön, T.B. (2020). Deep learning and system identification. *IFAC-PapersOnLine*, 53(2), 1175–1181.

Loshchilov, I. and Hutter, F. (2018). Decoupled weight decay regularization. In *International Conference on Learning Representations*.

Mavkov, B., Forgione, M., and Piga, D. (2020). Integrated neural networks for nonlinear continuous-time system identification. *IEEE Control Systems Letters*, 4(4), 851–856.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Piga, D., Forgione, M., and Mejari, M. (2021). Deep learning with transfer functions: new applications in system identification. *IFAC-PapersOnLine*, 54(7), 415–420.

Pillonetto, G., Aravkin, A., Gedon, D., Ljung, L., Ribeiro, A.H., and Schön, T.B. (2023). Deep networks for system identification: a survey. *arXiv preprint arXiv:2301.12832*.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. (2018). Improving language understanding by generative pre-training.

Rogge, N. and Rasul, K. (2022). Probabilistic time series forecasting with hugging face transformers. URL https://huggingface.co/blog/time-series-transformers#probabilistic-time-series-forecasting-with-%F0%9F%A4%97-transformers.

Rumelhart, D.E., Hinton, G.E., and Williams, R.J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088), 533–536.

Sayed, A.H. (2001). A framework for state-space estimation with uncertain models. *IEEE Transactions on Automatic Control*, 46(7), 998–1013.

Tsiamis, A., Matni, N., and Pappas, G. (2020). Sample complexity of Kalman filtering for unknown systems. In *Learning for Dynamics and Control*, 435–444. PMLR.

Tsiamis, A. and Pappas, G.J. (2022). Online learning of the kalman filter with logarithmic regret. *IEEE Transactions on Automatic Control*, 68(5), 2774–2789.