Transformers on Dynamical Systems - An Exploration of In-context Learning

Wentinn Liao

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-40
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-40.html

May 1, 2024

Transformers on Dynamical Systems - An Exploration of In-Context Learning

Saagar Sanghavi

December 2023

Abstract

Large Language Models (LLMs) have shown to be highly effective at performing in-context learning, where, given a prompt, the model can learn from the prompt and complete the sequence without needing to perform additional gradient steps or fine-tuning. In this project, we investigated the ability of Transformer models to perform in-context learning on linear dynamical systems. We first experimented with Transformers trained on a single system, where the task for evaluation was to filter noise on trajectories sampled from the same system. Then, we experimented with Transformers trained on multiple systems of the same type, where the task was to perform simultaneous system identification and filtering. This is still very much a work in progress, and I hope to continue to work on this in the coming weeks.

1 Introduction

Recent advancements in language modeling and the emergence of chatbots like OpenAI's ChatGPT, Google's Bard, and Anthropic's Claude have all proven to be highly effective at question answering, learning from mistakes, and responding to feedback. The underlying architecture behind all of these models is the Transformer Architecture [1], which since its introduction has revolutionized the field of artificial neural networks and become the go-to architecture for data-driven modeling.

Language modeling is the task of determining the probability of a given sequence of words in a sentence. Many NLP tasks, such as question answering, machine translation, reading comprehension, and summarization, can all be unified under the paradigm of language modeling.

Large Language Models (LLMs) such as GPT [2] have gained significant attention because of their remarkable ability to be used for a variety of Natural Language Processing tasks. Most notably, researchers have observed the emergent property of **in-context learning**, where large language models can learn from a prompt to fill in the blanks and complete the sequence on new, unknown sequences, without the need to take additional gradient steps or fine-tune the model.

While the abilities of language models like ChatGPT at tasks like question answering are shockingly good, further investigation of the training objective reveals that the task at hand is simply next-token prediction; the model is simply learning to predict the next token, given all the tokens in the prompt that it has seen so far. While in-context learning and adapting/responding to the prompt provided is not the original training objective, it is a curious emergent property of the transformer architecture and is incredibly powerful.

Bidirectional Language Models like BERT [3] are also useful to get contextualized token embeddings for use in downstream tasks. In the original implementation of BERT, an encoder-only transformer model was trained on the task of masked token recovery - a randomly chosen subset of 20% of the tokens were masked out, and the training objective was to recover the masked tokens and identify the correct order of two sequences. Later work, as in RoBERTA [4], improved the effectiveness of the BERT model by removing the sequence order prediction task as well as training on a much larger dataset and using a more robust training procedure.

In this project, we aimed to understand the abilities and limitations of LLMs in a toy environment, using this as a playground for interpretability. In particular, we considered dynamical systems with a Markovian hidden state and observations that are noisy linear projections of the hidden state, as described in 2.1. We first present a simple set of task environments that can be described by the dynamical system model. Next, we evaluate the effectiveness of these various baselines to perform filtering, as described in 2.7. Then, we compare the performance of the baselines to Transformer models, both the GPT-style transformer (decoder-only, using masked auto-regressive next-token prediction) and the BERT-style transformer (encoder-only, using true-sequence prediction from the sequence of noisy observations).

2 Background and Baselines

2.1 Dynamical Systems and Markovian Processes

We first introduce the problem setup: A discrete-time dynamical system with a continuous vector-valued state. We first considered simple linear systems of the form:

$$\vec{x}_{t+1} = A\vec{x}_t + B\vec{u}_t + \vec{v}_t \qquad \text{(Process Transition Model)}$$
$$\vec{y}_t = C\vec{x}_t + \vec{w}_t \qquad \text{(Observation Model)}$$

Here, we have the vectors:
- $\vec{x}_t \in \mathbb{R}^n$ is the hidden state of the system at timestep t
- $\vec{u}_t \in \mathbb{R}^m$ represents the control inputs at each timestep t
- $\vec{y}_t \in \mathbb{R}^p$ represents the observations at each timestep t

and the matrices:
- $A \in \mathbb{R}^{n\times n}$ represents the state transition dynamics from one timestep to the next
- $B \in \mathbb{R}^{n\times m}$ represents a linear transformation applied to the inputs
- $C \in \mathbb{R}^{p\times n}$ represents the linear observation model of the hidden state of the system

and the noise vectors:
- $\vec{v}_t \sim \mathcal{N}(\vec{0}, Q)$ is the **process noise**, a zero-mean Gaussian random vector with covariance matrix Q that affects the transitions
- $\vec{w} \sim \mathcal{N}(\vec{0}, R)$ is the **sensor noise**, a zero-mean Gaussian random vector with covariance matrix R that affects the observations.
- The process and sensor noises may be correlated, with $S = \mathbb{E}[\vec{v}_t \vec{w}_t^T]$ being the cross-covariance between noises. For simplicity, we considered cases where the noises were uncorrelated (i.e. $S = 0$).

The system described here is a continuous state-space generalization of a Hidden Markov Model (HMM) with a Markovian hidden state $\vec{x}_t$ and observations $\vec{y}_t$ that depend only on the current hidden state. For simplicity, we ignored the control inputs u_t and only considered a system that evolves according to the state transition dynamics and driving disturbances. We will use P_t to represent the estimate of the covariance matrix at timestep t.

2.2 Task Environments

We consider stable and semi-stable systems of various orders in this project. For simple debugging and visualization of the traces, we chose A to be a rotation matrix in $SO(2)$, i.e. a matrix of the form $\begin{bmatrix}\cos(\theta) & -\sin(\theta)\\ \sin(\theta) & \cos(\theta)\end{bmatrix}$, where θ is a small angle (usually $1°$).

We then tried training transformers to overfit to a single system, where the underlying stable system was defined by the transition dynamics A, and the observations were a sum of the system states (ie. $C = \begin{bmatrix}1 & 1 & ... & 1\end{bmatrix}$). Specifically, we chose the following matrices (Credit: Sultan Daniels):

$$A_{2\times 2} = \begin{bmatrix} 0.965 & 0.157 \\ -0.0317 & 0.802 \end{bmatrix} \quad A_{3\times 3} = \begin{bmatrix} 0.705 & -0.328 & 0.426 \\ 0.744 & 0.811 & 0.060 \\ 0.117 & 0.261 & 0.672 \end{bmatrix} \quad A_{4\times 4} = \begin{bmatrix} 0.581 & 0.150 & 0.170 & 0.00437 \\ 0.0898 & 0.641 & 0.243 & -0.134 \\ 0.0497 & -0.250 & 0.813 & 0.0952 \\ 0.127 & 0.224 & 0.325 & 0.324 \end{bmatrix}$$

$$A_{5\times 5} = \begin{bmatrix} 0.695 & 0.0787, -0.0496 & 0.0829 & 0.0955 & \\ -0.115 & 0.847 & 0.0364 & 0.221 & -0.0593 \\ 0.352 & 0.209 & 0.609 & -0.130 & -0.2596 \\ -0.00172 & 0.00736 & -0.151 & 0.685 & 0.279 \\ 0.383 & 0.218 & -0.0856 & -0.143 & 0.414 \end{bmatrix}$$

$$A_{6\times 6} = \begin{bmatrix} 0.669 & 0.0387 & -0.128 & -0.227 & -0.330 & 0.323 \\ 0.0260 & 0.540 & -0.0145 & -0.04015 & 0.299 & -0.255 \\ -0.00697 & -0.275 & 0.673 & -0.0381 & 0.119 & 0.0506 \\ 0.0776 & -0.0549 & 0.0558 & 0.311 & 0.0667 & 0.132 \\ 0.101 & -0.0117 & -0.166 & -0.00478 & 0.798 & -0.112 \\ 0.0843 & -0.0192 & -0.0831 & -0.0711 & 0.191 & 0.491 \end{bmatrix}$$

For training transformers that perform simultaneous system identification and filtering, we chose random eigenvalues within the unit circle. For systems of an even order, all eigenvalues came in complex-conjugate pairs. For systems of an odd order, there would be one purely real eigenvalue in addition to the pairs of eigenvalues that came as complex conjugates. We applied a random similarity transformation to a diagonal matrix of the eigenvalues to get our A matrix, where the entries of the random transformation matrix were drawn from a standard normal distribution.

The B matrix was set to 0 (or, equivalently, we had no inputs to the system) for simplicity.

The C matrix was usually simply the identity (so the observations $\vec{y}_t$ were simply a noisy reading of the true hidden state), but we also considered some cases where the observations were a lower-dimensional projection of the hidden state (i.e. the C matrix was wide).

The Q and R matrices were typically chosen to be a scaled versions of the identity matrix. The sensor noise was typically an order of magnitude larger than the process noise, since we wanted the system to evolve primarily according to its dynamics and not be overpowered by the process noise.

For debugging purposes, we would usually choose the initial state to be at $\begin{bmatrix}1 & 0\end{bmatrix}^T$ on the unit circle, then let the state continue to move around. For higher-order systems, we would choose the initial state to be a random vector in steady state (i.e. we would start at $\vec{0}$ and then burn 100 timesteps), or sample the initial state from the distribution $\mathcal{N}(\vec{0}, P_0)$ with identity covariance matrix $P_0 = I$.

2.3 System Identification via Least Squares

Consider the case where we fix the C matrix (observation model) to be the identity. The observations are the noisy versions of the true hidden state. System Identification (SysId) refers to the task of determining the transition dynamics (A matrix) of the system.

A simple approach to system identification is to treat the entries of the A matrix as unknown parameters. We can set up a data matrix D and a target vector $\vec{p}$ using the states of the trajectory that have been observed.

For example, in the simple case where $A \in \mathbb{R}^{2\times 2}$, we can label the entries:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

Say we have noisy readings of $\vec{x}_0 = \begin{bmatrix} x_0[0] \\ x_0[1] \end{bmatrix}$, $\vec{x}_1 = \begin{bmatrix} x_1[0] \\ x_1[1] \end{bmatrix}$, $\vec{x}_2 = \begin{bmatrix} x_2[0] \\ x_2[1] \end{bmatrix}$, $\vec{x}_3 = \begin{bmatrix} x_3[0] \\ x_3[1] \end{bmatrix}$, $\vec{x}_4 = \begin{bmatrix} x_4[0] \\ x_4[1] \end{bmatrix}$.

Using the transition dynamics $\vec{x}_{t+1} \approx A\vec{x}_t$ and treating the entries of the A matrix as unknown, we can set up an overdetermined system:

$$\begin{bmatrix} x_0[0] & x_0[1] & 0 & 0 \\ 0 & 0 & x_0[0] & x_0[1] \\ x_1[0] & x_1[1] & 0 & 0 \\ 0 & 0 & x_1[0] & x_1[1] \\ x_2[0] & x_2[1] & 0 & 0 \\ 0 & 0 & x_2[0] & x_2[1] \\ x_3[0] & x_3[1] & 0 & 0 \\ 0 & 0 & x_3[0] & x_3[1] \end{bmatrix} \begin{bmatrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{bmatrix} \approx \begin{bmatrix} x_1[0] \\ x_1[1] \\ x_2[0] \\ x_2[1] \\ x_3[0] \\ x_3[1] \\ x_4[0] \\ x_4[1] \end{bmatrix}$$

$$D\left[\vec{A}_{stack}\right] \approx \vec{p}$$

and then to solve for the entries of A using least squares:

$$\left[\vec{A}_{stack}\right] = (D^T D)^{-1} D^T \vec{p}$$

This approach chooses the A matrix to minimize the process noise as well as the sensor noise, and assumes that the observation model (C matrix) is fixed to the identity (i.e., $\vec{y}_t \approx \vec{x}_t$). As more data is observed, the estimate of the true A matrix becomes more accurate. We can generalize this to systems for arbitrary sizes of the A matrix.

2.4 Kalman Filtering

The Kalman Filter (KF) [5] is an algorithm to estimate the true state of a noisy linear (or linearized) system as described in 2.1, with known system matrices. The Kalman Filter works by recursively predicting an estimate of true state, then updating the prediction based on observations. We assume the initial state is normally distributed as $\vec{x}_0 \sim \mathcal{N}(\vec{0}, P_0)$, and our goal is to estimate the true $\vec{x}_t$. We call our estimate $\hat{\vec{x}}$ and want to express it as a function of observations $\vec{y}_0$, $\vec{y}_1$, ... $\vec{y}_{t-1}$. We want to minimize the energy of the covariance matrix P_t of the estimate, namely:

$$\min_{\hat{\vec{x}}_t} \mathbb{E}[(\vec{x}_t - \hat{\vec{x}}_t)(\vec{x}_t - \hat{\vec{x}}_t)^T] = P_t$$

We know the A, C matrices of the system, and we are given true observations $\vec{y}_0$, $\vec{y}_1$, ... $\vec{y}_{t-1}$. We assume a linear structure of the underlying system as given in 2.1. The Kalman Filter assumes that our observer follows the model

$$\begin{aligned}\hat{\vec{x}}_{t+1} &= A\hat{\vec{x}}_t + K_t(\vec{y}_t - \hat{\vec{y}}_t)\\ \hat{\vec{y}}_t &= C\hat{\vec{x}}_t\end{aligned}$$

where K_t is called the **Kalman Gain** and weights how much we should incorporate the **innovation**, $\vec{y}_t - \hat{\vec{y}}_t$, into our updated prediction.

2.4.1 The Forward Kalman Filter

The standard (Forward) Kalman filter provides an optimal Linear Quadratic Estimate (LQE) of the true system state, given the evidences up to the previous timestep. That is, we estimate:

$$\mathbb{E}[X_t | y_1, y_2, ...y_t]$$

The filter operates by alternating between "Predict" and "Update" steps. In the prediction step, we advance the estimate of the true state to the next timestep given the known dynamics, and incorporate the new uncertainty in the estimate of the covariance matrix. In the update step, we are given a new observation $\vec{y}_{t+1}$, recalculate our Kalman gains, and update the covariance matrix estimate as well as the estimate of the hidden state.

The Kalman Filter equations are given below.

Prediction Step:

$$\begin{aligned}\hat{\vec{x}}_{t+1|t} &= A\hat{\vec{x}}_t\\ \hat{P}_{t+1} &= AP_tA^T + Q\\ \hat{\vec{y}}_{t+1} &= C\hat{\vec{x}}_{t+1|t}\end{aligned}$$

Incorporate Evidence:

$$\begin{aligned}K_t &= \hat{P}_{t+1}C^T(C\hat{P}_{t+1}C^T + R)^{-1} && \text{Kalman Gain}\\ P_{t+1} &= (I - K_tC)\hat{P}_{t+1} && \text{Covariance Matrix}\\ \hat{\vec{x}}_{t+1} &= \hat{\vec{x}}_{t+1|t} + K_t(\vec{y}_{t+1} - \hat{\vec{y}}_{t+1})\end{aligned}$$

The Kalman Gains tend to converge rapidly to a steady-state value as the filter is repeatedly applied to inputs of a sequence. Note that the updates to the Kalman Gains K_t and Covariance Matrices $\hat{P}_t$ do not depend on the observations $\vec{y}_t$ and can thus be completed offline.

2.4.2 Backward Kalman Filter and Kalman Smoothing

The (forward) Kalman filter considers the evidence $\vec{y}_1$, $\vec{y}_2$, ... $\vec{y}_t$ seen so far, up to a timestep t. However, if we know **all** the observations $\vec{y}_1, \vec{y}_2$, ... $\vec{y}_T$ for the entire trajectory, we can obtain an even better estimate of the true hidden state. Denote the length of the trajectory as T. For the system described in 2.1, we can equivalently rewrite the system as:

$$\begin{aligned}\vec{x}_{t-1} &= A^{-1}\vec{x}_t + +\tilde{\vec{w}}_t\\ \vec{y}_t &= C\vec{x}_t + \vec{v}_t\end{aligned}$$

Running the Kalman Filter on the system expressed in this form, we can obtain $\mathbb{E}[X_t | y_T, y_{T-1}, ...y_t]$. We can effectively use the same algorithm as in 2.4.1, but replace A with A^{-1} and reverse the order of our observations $\vec{y}_t$. Then, we can obtain a smoothed prediction of the true hidden state by averaging the estimated state of both the forward and backward Kalman filter.

2.5 Optimization-based Trajectory Estimation

While the Forward-Backward Kalman filter uses the knowledge of the full trajectory to perform smoothing, the sequence of hidden states this recovers would double-count the uncertainty on the current state $\vec{x}_t$ and thus still not be the optimal sequence of hidden states given all the observations. To calculate the optimal sequence of hidden states, we can set up a least-squares optimization problem of minimizing the total energy of the noise of a system, where the energy associated with a noise vector is a quadratic form with the inverse of the covariance matrix. This corresponds to a maximum a posteriori (MAP) estimate of the true states, with Gaussian priors on the noise vectors. In other words, we want to solve:

$$\begin{aligned} &\underset{x_1, x_2, \ldots x_T}{\text{argmin}} \sum_{t=1}^{T} \left(\vec{w}_t^T Q^{-1} \vec{w}_t + \vec{v}_t^T R^{-1} \vec{v}_t \right) \\ &\text{where } \vec{w}_t = \vec{x}_{t+1} - A\vec{x}_t - B\vec{u}_t \\ &\qquad\quad \vec{v}_t = \vec{y}_t - C\vec{x}_t \end{aligned}$$

This problem is a convex Quadradratic Program (QP), and there exist many efficient algorithms to solve it. We used the CVXPY optimization package [6] with the Operator Splitting Quadratic Program (OSQP) solver [7] to solve the problem. The OSQP solver implements the Alternating Direction Method of Multipliers (ADMM) algorithm [8], which solves the problem efficiently.

2.6 Learning the Kalman Filter: A Recurrent Neural Network

Using the Kalman Filter equations in 2.4.1, we can rewrite the update rule for the Kalman Filter as:

$$\begin{aligned} \vec{x}_{t+1|t} &= A\vec{x}_t \\ \hat{\vec{y}}_{t+1} &= C\vec{x}_{t+1|t} \\ \vec{x}_{t+1} &= \vec{x}_{t+1|t} + K_t(\vec{y}_{t+1} - C\vec{x}_{t+1|t}) \end{aligned}$$

Notice that the Forward Kalman Filter update equations maintain a "hidden state" $\hat{\vec{x}}_t$ for each timestep that is weighted by a transition matrix $A' = (I - K_t C)A$ as it transitions to the next timestep, , and the previous (true) observation y_t is weighted by some matrix $G' = K_t$. The predicted next observation $\hat{\vec{y}}_{t+1}$ is a projection of the hidden state, i.e. $\hat{\vec{y}}_{t+1} = C'\hat{\vec{x}}_t$, where $C' = C$. If the system A, C, Q, R parameters are unknown (and thus the Kalman Gain K also cannot be calculated), we could try to learn the Kalman Filter's A', C', and G' parameters using the measured $\vec{y}_t$ vectors. This is the basic idea of a Recurrent Neural Network (RNN).

An RNN [9] is a commonly used Neural Network model for sequence modeling. Prior to the advent of transformers, they were commonly used for Language modeling and have also found use in state estimation for nonlinear systems [10] and time-series anlysis in general. An RNN maintains a hidden state x_t, and receives an input at each timestep. A linear transformation of the input is added to a linear transformation of the hidden state at each timestep. The weights that act on the inputs are shared across timesteps, and the weights on the hidden state transformation are shared across timesteps. We can also apply a linear transformation of the hidden state which turns the hidden state into an output for each timestep. During training, we feed true values $\vec{y}_t$ as the input to each timestep and calculate an MSELoss on the predicted $\hat{\vec{y}}$ compared to the true values, while during inference we generate predictions in an autoregressive fashion, using the $\hat{\vec{y}}_t$ from the current timestep oncatenated with the control input $\vec{u}_t$ as the input for the next timestep.

For modeling sequences that are emitted from a linear dynamical system, we need to consider autoregressive generation on the decoder-side. We provide the observations $\vec{y}_t$ as the inputs to the transformer and we have an identical situation to the Kalman Filter, where the matrix $A' = (I - KC)A$ represents the dynamics Jacobian from one timestep to the next, and the matrix $G' = AK$ represents the weights on the observations $\vec{y}_t$. Finally, we have the matrix $C' = C$ that acts on the hidden state to produce a predicted output $\hat{\vec{y}}_t$ for each timestep.

With the simple Kalman filter, the A, C, Q, R matrices are given, and the Kalman gain K can be calculated analytically. However, it is also possible to learn the Kalman Filter parameters by treating A', G', C' as learnable weight matrices, compute the total loss of the predicted measurements compared to the actual measurements $\sum_{t=1}^{T}(\hat{\vec{y}}_t - \vec{y}_t)$ for a given sequence, pass gradients through, and update the weight matrices using gradient descent.

2.7 Summary of Baselines

For the next-token prediction task, we have the following baselines that we can compare to the GPT-style Decoder-only Transformer Model:

- ZERO: Simply predict 0 at each timestep
- PREV: Predict the previous observation at each timestep
- KF: Use the closed-form analytical Kalman Filter solution, with the known A, C, Q, and R matrices
- IDKF: Perform system identification (assuming C to be the identity matrix) to recover the A matrix, then update the Kalman Filter parameters and perform a one-step prediction.
- LearnedKF: Learn the Kalman Filter Parameters, as described in 2.6. This is the strong baseline, since it does not require knowledge of the A, C, Q, R matrices. I was unable to get this part of the project working, but we brought Wentinn Liao to work on this problem, and his preliminary results are in [11]

For the true-sequence recovery task, we have the following baselines that we can compare to the BERT-style Encoder-only Transformer Model:

- ZERO: Simply predict 0 for each timestep
- LERP: Perform Linear interpolation of the previous and next timesteps
- KF_SMOOTH: Average of the forward and backward Kalman Filters, as described in 2.4.2, using the known first and last states as initial conditions for the Kalman Filter.
- LS_OPT: Use optimization-based Trajectory Estimation to minimize the energy of the noise using the Least Squares approach described in 2.5 to recover a MAP estimate of the true trajectory. This requires knowledge of the A, C, Q, and R matrices.

3 Transformer Models

Transformer Models have revolutionized the field of deep learning. While originally proposed for Language Sequence Modeling, they have recently found use in other tasks including computer vision [12] as well as audio and multimodal processing. The basic building block uses stacked layers of self-attention, which is effectively "queryable softmax pooling" that allows the network architecture to learn long-range dependencies between tokens.

Transformers use the basic mechanism of query-key-value attention in order to learn long-range dependencies. While originally proposed as a mechanism that could operate on RNNs, the seminal work of Attention is All You Need [1] showed that removing the RNN backbone and incorporating positional encoding was enough for the Transformer block to be used in neural networks on its own.

The original Transformer approach used both an encoder and decoder for sequence-to-sequence tasks. The encoder side performs unmasked self-attention, while the decoder side uses cross-attention (i.e. queries from the decoder-side attend to keys and values from the encoder side) and masked decoding to prevent lookups into the future. Later advances in laugage modeling found that Encoder-only and Decoder-only models can also perform well when pretrained for certain tasks. In particular, BERT [3] utilized an encoder-only architecture for masked token recovery, and GPT [2] utilized a decoder-only architecture for next-token prediction.

3.1 Decoder-only Transformer: GPT and Next-Token Prediction

One set of transformers we trained were GPT-style decoder-only transformers, where the target was to predict the next (noisy) observation given the sequence of noisy measurements and control inputs so far. We applied causal masking to prevent future lookups and prevent learning the identity map. We use models in the GPT-2 family [2]. The model applied teacher-forcing during training (i.e. the true y_t values were fed), but during inference, the new tokens were generated in an autoregressive manner (i.e. feeding in the previous $\hat{y}_t$ at each timestep).

The GPT-2 Model, as introduced in [2], shows remarkable capability on a wide variety of NLP tasks. The basis of the approach taken is language modeling, which frames the task as estimating the distribution of tokens $p(x) = p(x_1, x_2, ...x_n) = \prod_{i=1}^{n} p(s_n|s_1, s_2, ...s_{n-1})$, using the samples $(s_1, s_2, ...s_n)$. By estimating the joint distribution as a product of conditional probability distributions, the sampling is tractable.

In the original paper, the GPT2Model was trained on a dataset that was generated by scraping web pages from Reddit that received at least 3 karma points—this way, only web pages that were interesting or usefully informative were used in the common crawl. The input tokenization scheme used byte-pair encoding (BPE) scheme as a practical middle ground between character- and word-level inputs. The authors created models of four different sizes and evaluated the performance of the models on a variety of tasks, namely Language Modeling on the WebText, Children's book test, the Lambada dataset, the Winograd Schema Challenge, Reading Comprehension, Summarization, Translation, and Question-Answering. They found the language model to be highly effective at several of the tasks.

In our experiments, we generated new training trajectories on the fly from the systems we were trying to model, and we apply a linear projection before feeding the data into the model. There was no need for any tokenization scheme in this process since we were working directly with a sequence of measurement vectors.

3.2 Encoder-only Transformer: BERT-style True Sequence Recovery

Another set of transformers we trained were the BERT-style Encoder-only Transformers, where the target was to recover the true state trajectory, and the inputs were the sequence of noisy measurements at each timestep.

The BERT model [3] was introduced in 2018 and used an Encoder-only Transformer architecture, trained on the task of masked token recovery as well as the surrogate task of next sequence classification. The input to the model would consist of two related sequences, for example a (question, answer) pair or a (Foreign, English) translation pair, where a random subset of the tokens were replaced with a [MASK] token and half the times the order of the sequences would be swapped. BERT used an embedding scheme called WordPiece, which maps each English word to an integer code, for words in a vocabulary of size 30K. Any token that didn't appear in the vocabulary would be replaced by an [UNK] token for unknown characters. BERT was fine-tuned on several downstream NLP tasks, including the General Language Understanding Evaluation (GLUE) benchmark, Stanford Question-Answering Dataset (SQuAD), and Situations with Adversarial Generations (SWAG) dataset. Furthermore, the authors performed several ablation studies, looking at the effect of pretraining tasks, the effect of model size, the feature-based approach with BERT. The contextual embeddings from BERT appeared to be highly effective on a variety of tasks including text classification, named entity recognition, question answering, language understanding, and machine translation. The effect of BERT on the NLP community was quite profound, and it later became the basis for various other SOTA lanugage models and pretrained embeddings. Researchers and developers fine-tune BERT for specific tasks, often using transfer learning, where the model is pretrained on a large corpus of text data and then adapted to a specific NLP task.

For our task, because we were trying to recover the true trajectories from the noisy measurements, we do not need to perform any masking and can simply use the encoder-side transformer as a sequence model for all the tokens.

3.3 Large Language Models, Pretraining and Fine-tuning

Large Language Models (LLMs) are neural network models with a large number of parameters, trained on large datasets. The training objective is usually either next-token prediction or masked token recovery. The training is typically distributed across a large cluster of GPUs and requires several feats of engineering to distribute and optimize training and inference. While classical NLP techniques tended to use different techniques for different tasks like question answering, named entity recognition, sentiment analysis, translation, etc., modern approaches treat Language Modeling as the ultimate task, and all other tasks can be accomplished through prompting the LLM. Recently, LLMs have gained widespread popularity for their success and state-of-the-art performance.

Pretraining refers to the process of training an LLM on a large corpus of unlabeled data that can be used for a variety of downstream tasks, while fine-tuning refers to the process of performing additional gradient steps for a specific downstream task. During pretraining, the model learns to predict the next word in a sentence, given the preceding words. This process results in the model acquiring knowledge of the grammar, vocabulary, syntax, and factual information present in the text. The objective of pretraining is to allow the model to gain a basic understanding of the language and develop generation abilities.

Fine-tuning involves training the pretrained model on the smaller dataset while adjusting its parameters to perform well on a specific task. The key advantage of fine-tuning is that it allows the pretrained model to leverage the general language understanding it learned during pretraining while adapting to the specific nuances of the task or domain during fine-tuning.

3.4 In-context Learning and Adaptation

Humans commonly use knowledge of the information seen so far in decision-making. Similarly, large language models can use the context seen so far in order to predict the next token. LLMs have shown to be successful in performing *prompting*, where, given a few training examples, the model can quickly pick up the pattern and fill in the answer on new examples. For example, given the prompt "The capital of France is", the model will output "Paris" with high probability.

Unlike fine-tuning, prompting does not require performing any additional gradient steps or training on the model; rather, it simply provides right sequence of inputs so that model is in the right "frame of reference" to predict the correct answer.

Prompting appears to be an emergent property of Transformer models that is not fully understood; after all, transformer models are typically trained on the task of next-token prediction or masked token recovery, so it is quite interesting that the model can adapt to different downstream tasks by simply providing different input sequences.

This idea of in-context learning has been explored across many domains. However, it is unclear how models perform in-context learning, and the limitations of ICL.

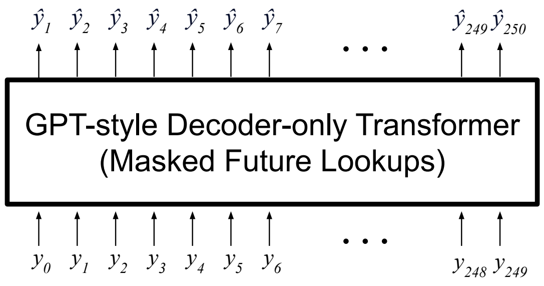
Recent work on what transformers can learn in-context [13] explores the different types of functions that GPT-style decoder-only transformers can learn, focusing on sparse linear functions, two-layer neural networks, and decision trees. This paper found that transformers are capable of in-context learning linear functions with performance similar to that of least squares. Simpler baselines of averaging and 3-nearest neighbors are also able to in-context learn and achieve non-trivial error, but their performance does not match the effectiveness of the transformers—this indicates that transformers are actually able to learn a more complex algorithm than the trivial baselines.

4 Experimental Setup

4.1 Model Architectures for GPT-style Decoder-only Transformer

We used decoder-only style transformers from the GPT2 family [2], implemented in HuggingFace [14]. We consider models of the following sizes:

Model	n_embd	n_layer	n_head
Small	32	3	2
Medium	64	6	4
Large	128	12	8


$\hat{y}_1$ $\hat{y}_2$ $\hat{y}_3$ $\hat{y}_4$ $\hat{y}_5$ $\hat{y}_6$ $\hat{y}_7$ ... $\hat{y}_{249}$ $\hat{y}_{250}$
GPT-style Decoder-only Transformer
(Masked Future Lookups)
y_0 y_1 y_2 y_3 y_4 y_5 y_6 ... y_{248} y_{249}

The diagram to the right shows the training procedure of the task of next-token prediction, i.e. predicting the next observation $\vec{y}_t$ given the sequence of observations so far $\vec{y}_0, \vec{y}_1, \cdots, \vec{y}_{t-1}$.

Since training data is generated on-the-fly, overfitting is not a concern and thus we do not perform dropout (i.e. we set the dropout probability to 0). To convert the trajectories into embeddings that are fed into the Transformer, we apply a learned linear readin layer. We also apply a learned linear readout layer to the output generated by the Transformer to recover the next token (we initially experimented with weight-sharing between the embedding end readout layers, but this was found to be ineffective and thus we choose to learn the weights separately).

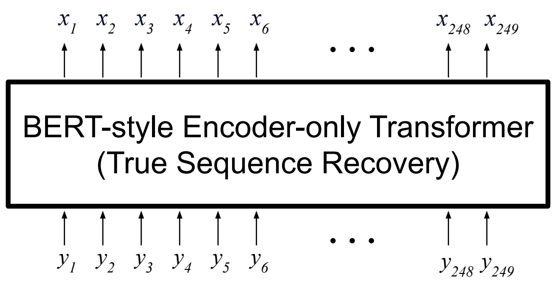
4.2 Work in Progress: Model Architectures for BERT-style Encoder-only Transformer

We can also train an Encoder-only Transformer as described in [1], which is implemented in PyTorch [15] as TransformerEncoder. We consider models of the following sizes:

Model	n_embd	n_layer	n_head
Small	32	3	2
Medium	64	6	4
Large	128	12	8

The diagram to the right shows the training procedure for the task of true-sequence recovery, i.e. recovering the true sequence $\vec{x}_0, \vec{x}_1, ...\vec{x}_t$ given the sequence of observations $\vec{y}_0, \vec{y}_1, ...\vec{y}_t$.

We have not had success with this so far, but we are working on getting this working before the ISIT deadline.


x_1 x_2 x_3 x_4 x_5 x_6 . . . x_{248} x_{249}
BERT-style Encoder-only Transformer
(True Sequence Recovery)
y_1 y_2 y_3 y_4 y_5 y_6 . . . y_{248} y_{249}

Again, since the training data is generated on-the-fly, we do not perform dropout. As with the GPT-style models, we apply a learned linear readin layer before the TransformerEncoder backbone and also apply a learned linear readout layer to the outputs of the Transformer.

4.3 Computational Resources

All Saagar's models were trained locally on a MacBook Pro. Wentinn and Sultan Trained their models on Google Colab with GPU.

5 Experimental Results

The graphs below (credit to Wentinn Liao and Sultan Daniels) show the relative errors of the Time-series Transformer (which only works with univariate inputs), that aims to use the Transformer to perform filtering of noise trained on a single system. More details can be found in [11].

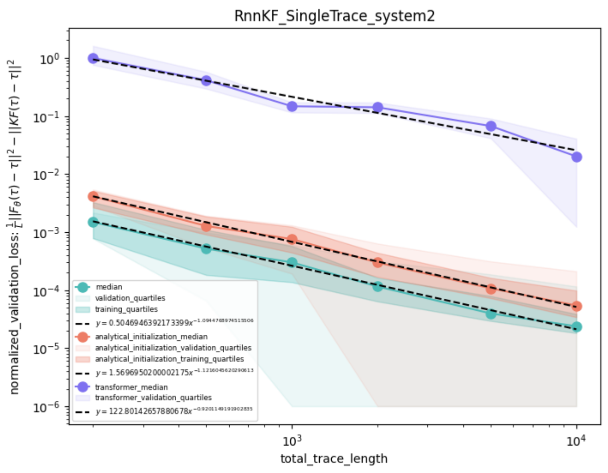


Figure 1: Transformers vs the LearnedKF on the order-2 system described above.

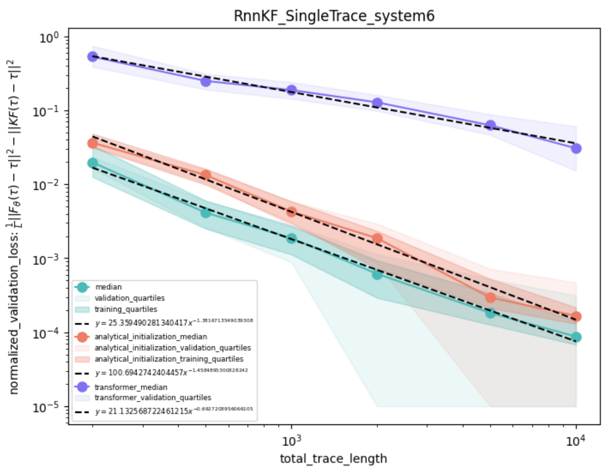


Figure 2: Transformers vs the LearnedKF on the order-6 system described above.

We can see that the Transformer performs well, though not optimally, on the task of next-token prediction.

My work aims to generalize Sultan's results to work on arbitrary sized observations, i.e. systems where we have more than a single variable output (i.e., the C matrix is not a row vector). I trained my transformers on the same systems, and obtained the following results for the overall errors of the trajectories compared to the optimal Kalman Filter. The graphs below show plots of the median overall error of the trajectory using the small GPT2 decoder-only Transformer Model as compared to other methods.

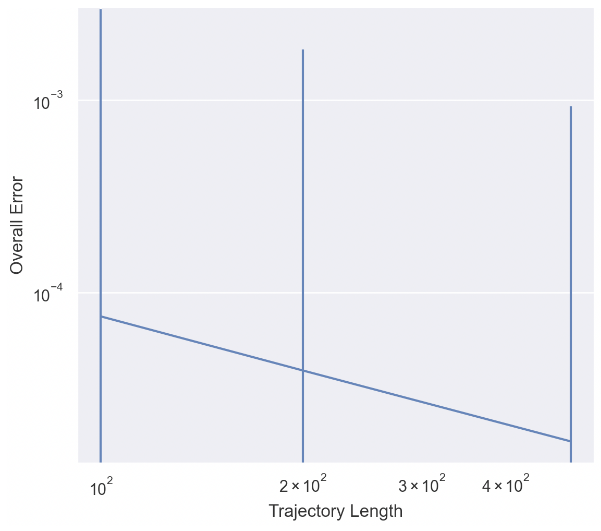


Figure 3: Overall Error of Transformer Trace vs. Trajectory Length. Mean taken over 100 traces. Error Bars show 1SD above and below the mean.

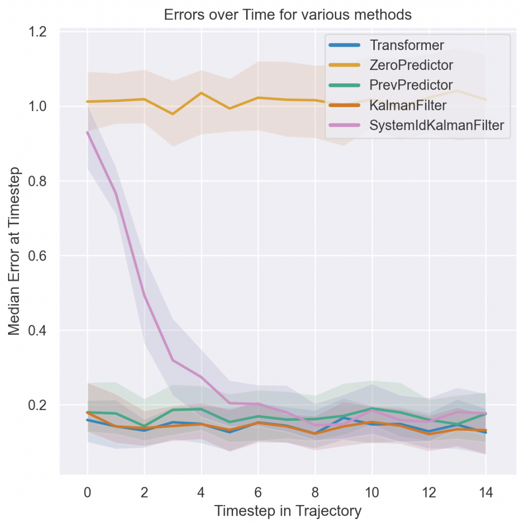


Figure 4: Median Error (IQR shaded) vs Timestep in Trajectory. This is averaged over 100 trajectories.

This graph on the right confirms the scaling law observed by Sultan, however the errors are an order of magnitude off. We can see that the GPT-style model performs well, though not optimally, on the filtering task. This merits further investigation, since Sultan's work used the Time Series Transformer as a black box, while I used the GPT2 Transformer from Huggingface as a black box.

6 Future Work

So far, we have had some success in training GPT-style models on a single system to perform filtering via the task of next-token prediction. In the future, I hope to also train BERT-style models to perform true-sequence recovery, as described in the previous section. We also want to train models to perform simultaneous system identification and filtering, which would require training much larger Transformer models on traces from different systems. This is analogous to in-context learning in practice, since the model needs to be able to adapt to different downstream tasks and generalize (in this case, the "tasks" correspond to different systems, and the adaptation corresponds to filtering).

Sultan, Wentinn, and I have all been working on separate repositories. My goal is to create a unified, clean testing framework that allows a user to specify whether to train a single random system of order n, a single system specified by a matrix, or general order-n system. I hope to work with Sultan and Wentinn to integrate the changes into a single testing framework and be able to get clean, apples-to-apples comparisons on the results.

I've learned a lot from the people around me, both in terms of the tenacity and persistence that it takes to do research. I hope to keep that with me as a continue working on this project.

My work can be found here: https://github.com/ssanghavi404/dynamical-systems This is still a work in progress. In the next week, I hope to integrate my changes with Sultan's work.

References

[1] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[2] Alec Radford et al. *Language models are unsupervised multitask learners*. 2019. URL: https://www.semanticscholar.org/paper/Language-Models-are-Unsupervised-Multitask-Learners-Radford-Wu/9405cc0d6169988371b2755e573cc28650d14dfe (visited on 01/06/2023).

[3] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[4] Yinhan Liu et al. “RoBERTa: (A) Robustly Optimized BERT Pretraining Approach”. In: *CoRR* abs/1907.11692 (2019). arXiv: `1907.11692`. URL: `http://arxiv.org/abs/1907.11692`.

[5] Rudolph Emil Kalman. “A New Approach to Linear Filtering and Prediction Problems”. In: *Transactions of the ASME–Journal of Basic Engineering* 82 (1960).

[6] Steven Diamond and Stephen Boyd. “CVXPY: A Python-embedded modeling language for convex optimization”. In: *Journal of Machine Learning Research* 17.83 (2016), pp. 1–5.

[7] B. Stellato et al. “OSQP: an operator splitting solver for quadratic programs”. In: *Mathematical Programming Computation* 12.4 (2020), pp. 637–672. DOI: `10.1007/s12532-020-00179-2`. URL: `https://doi.org/10.1007/s12532-020-00179-2`.

[8] Stephen Boyd. *Distributed optimization and statistical learning via the Alternating Direction Method of Multipliers*. 2011. URL: `https://web.stanford.edu/~boyd/papers/pdf/admm_distr_stats.pdf`.

[9] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning internal representations by error propagation: Tech. rep. ICS 8504.” In: (Sept 1985).

[10] Mohammed S. Alhajeri et al. “Machine-learning-based state estimation and predictive control of nonlinear processes”. In: *Chemical Engineering Research and Design* 167 (2021), pp. 268–280. ISSN: 0263-8762. DOI: `https://doi.org/10.1016/j.cherd.2021.01.009`. URL: `https://www.sciencedirect.com/science/article/pii/S0263876221000095`.

[11] *Learning the Kalman Filter*. 2023.

[12] Alexey Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *CoRR* abs/2010.11929 (2020). arXiv: `2010.11929`. URL: `https://arxiv.org/abs/2010.11929`.

[13] Shivam Garg et al. “What Can Transformers Learn In-Context? A Case Study of Simple Function Classes”. In: *arxiv:pre-print* (2023).

[14] Thomas Wolf et al. *HuggingFace’s Transformers: State-of-the-art Natural Language Processing*. 2020. arXiv: `1910.03771 [cs.CL]`.

[15] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: `1912.01703 [cs.LG]`.