# Supplementary Material

**Outline.** Supplementary material is outlined as follows. Section A discusses the impact of the proposed model. Section B details the proof of the Fourier feature kernels. Section C gives the derivation of the training loss of an NTK. Section D provides details of the training pipeline and the model architecture. Finally, Section E assembles the additional experimental results. These include:

(1) visualization of encoded states estimated from the self-attention network of ALPS and its attention maps in the pendulum task,

(2) visualization of reconstructed observations and prediction of system parameters for each task, and

(3) analysis of self-attention networks in predicting states from observations.

## A Impact of ALPS

Many control systems such as self-driving cars and autonomous robots are complex. In order to deal with this complexity, researchers are increasingly using reinforcement learning (RL) to design control policies. One approach is to design a *model-based* RL algorithm, in which we first learn a model of the system dynamics from trajectory data, and then use this model to learn or synthesize a policy. Compared to the approach that uses neural networks to model the dynamics, ALPS can exploit prior knowledge about the system dynamics to have better learning efficiency and generalization. In addition, using continuous-time system dynamics allows us to better capture the behavior of the system, and incorporate other control properties into the model such as stability and controllability of the system.

One assumption of ALPS is that we need to have prior knowledge of the system. This assumption is reasonable since in many practical applications, system designers can use the law of physics to design models. For example, we know Newton's laws of motion are still true on Mars, but its gravitational constant is different. However, for some systems such as contact dynamics, we do not have good knowledge about how to design models. We make assumptions that ALPS requires knowing the equation of the system dynamics and having a differentiable solver for it. These assumptions are too restrictive to make this useful in practice. A potential way to overcome this is to develop a hybrid approach, in which we use positions and velocities that follow the law of physics as latent state representations, followed by network networks that capture complex dynamics to compensate for unmodeled parts of the system.

This raises several questions: What kind of neural architecture is needed to capture the dynamics of the system? How to applied ALPS in more complicated settings or unseen objects during testing? We believe this paper will encourage future work to develop rigorous design and analysis tools for learning system dynamics.

## B Proof of the Kernel Functions

We now derive the kernel of the Fourier feature mapping. We review the setup here. Consider a state trajectory $\boldsymbol{v} = [x_0, x_1, \dots, x_{\tau-1}]^T$, where here we consider a 1D case for a scalar state $x \in \mathbb{R}$ although it can be easily extended to a vector state $\boldsymbol{x} \in \mathbb{R}^n$. The feature maps of the Fourier feature mapping, their magnitudes, and their phases are

$$\phi_{\mathrm{DFT}}(\boldsymbol{v}) = [X_0, \dots, X_k, \dots, X_{\tau-1}]^T \in \mathbb{R}^\tau, X_k = \sum_{j=0}^{\tau-1} x_j \Big[ \cos\Big(\frac{2\pi}{\tau} kj\Big) - i \sin\Big(\frac{2\pi}{\tau} kj\Big)\Big];$$

$$\phi_{\mathrm{MAG}}(\boldsymbol{v}) = [|X_0|, \dots, |X_k|, \dots, |X_{\tau-1}|]^T \in \mathbb{R}^\tau;$$

$$\phi_{\mathrm{PHA}}(\boldsymbol{v}) = [\arg(X_0), \dots, \arg(X_k), \dots, \arg(X_{\tau-1})]^T \in \mathbb{R}^\tau.$$

Set $\boldsymbol{C}_k = \left[\cos\left(\frac{2\pi}{\tau}ki\right) - \sin\left(\frac{2\pi}{\tau}kj\right)\right] \in \mathbb{R}^{\tau\times\tau}$, then the kernel functions of these mappings are

$$
\begin{aligned}
k_{\mathrm{DFT}}(\boldsymbol{v}_1, \boldsymbol{v}_2) =& \sum_{k=0}^{\tau-1} \boldsymbol{v}_1^T \boldsymbol{C}_k \boldsymbol{v}_2; \\
k_{\mathrm{MAG}}(\boldsymbol{v}_1, \boldsymbol{v}_2) =& \sum_{k=0}^{\tau-1} \sqrt{\boldsymbol{v}_1^T \boldsymbol{C}_k \boldsymbol{v}_1 \boldsymbol{v}_2^T \boldsymbol{C}_k \boldsymbol{v}_2}; \\
k_{\mathrm{PHA}}(\boldsymbol{v}_1, \boldsymbol{v}_2) =& \phi_{\mathrm{PHA}}(\boldsymbol{v}_1)^T \phi_{\mathrm{PHA}}(\boldsymbol{v}_2).
\end{aligned}
$$

*Proof:*

**(1)** $k_{\mathrm{DFT}}(\boldsymbol{v}_1, \boldsymbol{v}_2)$ : Let $\boldsymbol{v}_1 = \begin{bmatrix} x_{1,1}, \\ \vdots, \\ x_{\tau-1,1} \end{bmatrix}$ and $\boldsymbol{v}_2 = \begin{bmatrix} x_{1,2}, \\ \vdots, \\ x_{\tau-1,2} \end{bmatrix}$. Then

$$
\begin{aligned}
& k_{\mathrm{DFT}}(\boldsymbol{v}_1, \boldsymbol{v}_2) \\
& \quad = \phi_{\mathrm{DFT}}(\boldsymbol{v}_1)^T \phi_{\mathrm{DFT}}(\boldsymbol{v}_2) \\
& \quad = \left[\sum_{j=0}^{\tau-1} x_{j,1} \cos\left(\frac{2\pi}{\tau} j \cdot k\right), \sum_{j=0}^{\tau-1} x_{j,1} \sin\left(\frac{2\pi}{\tau} j \cdot k\right)\right]_k \begin{bmatrix} \sum_{j=0}^{\tau-1} x_{j,2} \cos\left(\frac{2\pi}{\tau} j \cdot k\right) \\ \sum_{j=0}^{\tau-1} x_{j,2} \sin\left(\frac{2\pi}{\tau} j \cdot k\right) \end{bmatrix}_k \\
& \quad = \sum_{k=0}^{\tau-1} \left( \boldsymbol{v}_1^T \begin{bmatrix} \cos\left(\frac{2\pi}{\tau} \cdot 1 \cdot k\right) \\ \vdots \\ \cos\left(\frac{2\pi}{\tau} \cdot (\tau-1) \cdot k\right) \end{bmatrix} \boldsymbol{v}_2^T \begin{bmatrix} \cos\left(\frac{2\pi}{\tau} \cdot 1 \cdot k\right) \\ \vdots \\ \cos\left(\frac{2\pi}{\tau} \cdot (\tau-1) \cdot k\right) \end{bmatrix} \right. \\
& \qquad \left. + \boldsymbol{v}_1^T \begin{bmatrix} \sin\left(\frac{2\pi}{\tau} \cdot 1 \cdot k\right) \\ \vdots \\ \sin\left(\frac{2\pi}{\tau} \cdot (\tau-1) \cdot k\right) \end{bmatrix} \boldsymbol{v}_2^T \begin{bmatrix} \sin\left(\frac{2\pi}{\tau} \cdot 1 \cdot k\right) \\ \vdots \\ \sin\left(\frac{2\pi}{\tau} \cdot (\tau-1) \cdot k\right) \end{bmatrix} \right) \\
& \quad = \sum_{k=0}^{\tau-1} \boldsymbol{v}_1^T \left( \begin{bmatrix} \cos\left(\frac{2\pi}{\tau} \cdot 1 \cdot k\right) \\ \vdots \\ \cos\left(\frac{2\pi}{\tau} \cdot (\tau-1) \cdot k\right) \end{bmatrix} \left[\cos\left(\frac{2\pi}{\tau} \cdot 1 \cdot k\right), \ldots, \cos\left(\frac{2\pi}{\tau} \cdot (\tau-1) \cdot k\right)\right] \right. \\
& \qquad \left. + \begin{bmatrix} \sin\left(\frac{2\pi}{\tau} \cdot 1 \cdot k\right) \\ \vdots \\ \sin\left(\frac{2\pi}{\tau} \cdot (\tau-1) \cdot k\right) \end{bmatrix} \left[\sin\left(\frac{2\pi}{\tau} \cdot 1 \cdot k\right), \ldots, \sin\left(\frac{2\pi}{\tau} \cdot (\tau-1) \cdot k\right)\right] \right) \boldsymbol{v}_2 \\
& \quad = \sum_{k=0}^{\tau-1} \boldsymbol{v}_1^T \boldsymbol{C}_k \boldsymbol{v}_2,
\end{aligned}
$$

which completes the proof of $k_{\mathrm{DFT}}(\boldsymbol{v}_1, \boldsymbol{v}_2)$.

**(2)** $k_{\mathrm{MAG}}(\boldsymbol{v}_1, \boldsymbol{v}_2)$ : Let $\boldsymbol{c}_k = \begin{bmatrix} \cos\left(\frac{2\pi}{\tau} \cdot k \cdot 0\right) \\ \vdots \\ \cos\left(\frac{2\pi}{\tau} \cdot k \cdot (\tau-1)\right) \end{bmatrix}$ and $\boldsymbol{s}_k = \begin{bmatrix} \sin\left(\frac{2\pi}{\tau} \cdot k \cdot 0\right) \\ \vdots \\ \sin\left(\frac{2\pi}{\tau} \cdot k \cdot (\tau-1)\right) \end{bmatrix}$. Then

$$
\begin{aligned}
&k_{\mathrm{MAG}}(\boldsymbol{v}_1, \boldsymbol{v}_2) \\
&\quad = \phi_{\mathrm{MAG}}(\boldsymbol{v}_1)^T \phi_{\mathrm{MAG}}(\boldsymbol{v}_2) \\
&\quad = \left[ \sqrt{\left( \sum_{j=0}^{\tau-1} x_{j,1} \cos\left(\tfrac{2\pi}{\tau} j \cdot k\right) \right)^2 + \left( \sum_{j=0}^{\tau-1} x_{j,1} \sin\left(\tfrac{2\pi}{\tau} j \cdot k\right) \right)^2} \right]_k^T \\
&\qquad\qquad \left[ \sqrt{\left( \sum_{j=0}^{\tau-1} x_{j,2} \cos\left(\tfrac{2\pi}{\tau} j \cdot k\right) \right)^2 + \left( \sum_{j=0}^{\tau-1} x_{j,2} \sin\left(\tfrac{2\pi}{\tau} j \cdot k\right) \right)^2} \right]_k \\
&\quad = \sum_{k=0}^{\tau-1} \left( \sqrt{\left(\boldsymbol{v}_1^T \boldsymbol{c}_k\right)^2 + \left(\boldsymbol{v}_1^T \boldsymbol{s}_k\right)^2} \sqrt{\left(\boldsymbol{v}_2^T \boldsymbol{c}_k\right)^2 + \left(\boldsymbol{v}_2^T \boldsymbol{s}_k\right)^2} \right) \\
&\quad = \sum_{k=0}^{\tau-1} \sqrt{\left(\boldsymbol{v}_1^T \boldsymbol{s}_k \boldsymbol{c}_k^T \boldsymbol{v}_2\right)^2 + \left(\boldsymbol{v}_1^T \boldsymbol{c}_k \boldsymbol{s}_k^T \boldsymbol{v}_2\right)^2 + \left(\boldsymbol{v}_1^T \boldsymbol{c}_k \boldsymbol{c}_k^T \boldsymbol{v}_2\right)^2 + \left(\boldsymbol{v}_1^T \boldsymbol{s}_k \boldsymbol{s}_k^T \boldsymbol{s}_2\right)^2}. \qquad (2)
\end{aligned}
$$

Set $a_k = \boldsymbol{v}_1^T \boldsymbol{s}_k, b_k = \boldsymbol{v}_2^T \boldsymbol{c}_k, c_k = \boldsymbol{v}_1^T \boldsymbol{c}_k$, and $d_k = \boldsymbol{v}_2^T \boldsymbol{s}_k$. Then

$$
\begin{aligned}
(2) &= \sum_{k=0}^{\tau-1} \sqrt{\left(a_k b_k\right)^2 + \left(c_k d_k\right)^2 + \left(b_k c_k\right)^2 + \left(a_k d_k\right)^2} \\
&= \sum_{k=0}^{\tau-1} \sqrt{(a_k^2 + c_k^2)(b_k^2 + d_k^2)} \\
&= \sum_{k=0}^{\tau-1} \sqrt{\left(\boldsymbol{v}_1^T \boldsymbol{s}_k \boldsymbol{s}_k^T \boldsymbol{v}_1 + \boldsymbol{v}_1^T \boldsymbol{c}_k \boldsymbol{c}_k^T \boldsymbol{v}_1\right)^2 + \left(\boldsymbol{v}_2^T \boldsymbol{c}_k \boldsymbol{c}_k^T \boldsymbol{v}_2 + \boldsymbol{v}_2^T \boldsymbol{s}_k \boldsymbol{s}_k^T \boldsymbol{v}_2\right)^2} \\
&= \sum_{k=0}^{\tau-1} \sqrt{\left(\boldsymbol{v}_1^T (\boldsymbol{s}_k \boldsymbol{s}_k^T + \boldsymbol{c}_k \boldsymbol{c}_k^T) \boldsymbol{v}_1\right)^2 + \left(\boldsymbol{v}_2^T (\boldsymbol{c}_k \boldsymbol{c}_k^T + \boldsymbol{s}_k \boldsymbol{s}_k^T) \boldsymbol{v}_2\right)^2} \\
&= \sum_{k=0}^{\tau-1} \sqrt{\boldsymbol{v}_1^T \boldsymbol{C}_k \boldsymbol{v}_1 \boldsymbol{v}_2^T \boldsymbol{C}_k \boldsymbol{v}_2},
\end{aligned}
$$

which completes the proof of $k_{\mathrm{MAG}}(\boldsymbol{v}_1, \boldsymbol{v}_2)$. □

# C Derivation of the Training Loss of NTK

We now provide more details on the derivation of the training error of a neural network after $t$ times update. To be complete, we review the notations here. Let $\boldsymbol{K} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{U}^T$ denote the eigendecomposition of the kernel matrix $\boldsymbol{K}$ which must be positive semidefinite (PSD). Here $\boldsymbol{U}$ is an orthogonal matrix and $\boldsymbol{\Sigma}$ is a diagonal matrix whose entries are the nonnegative eigenvalues ordered

by magnitude: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_m \geq 0$. So the training error in terms of $L_2$ norm $\|\cdot\|_2^2$ is

$$\begin{aligned}
\left\|\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}\right\|_2^2 &= \left\| \begin{bmatrix} \boldsymbol{y}^T(\boldsymbol{I} - e^{-\eta \boldsymbol{K} t})\boldsymbol{K}^{-1}k(\boldsymbol{v}_1) \\ \vdots \\ \boldsymbol{y}^T(\boldsymbol{I} - e^{-\eta \boldsymbol{K} t})\boldsymbol{K}^{-1}k(\boldsymbol{v}_m) \end{bmatrix}_{m\times 1} - \boldsymbol{y} \right\|_2^2 \\
&= \left\|\boldsymbol{K}\boldsymbol{K}^{-1}(\boldsymbol{I} - e^{-\eta \boldsymbol{K} t})\boldsymbol{y} - \boldsymbol{y}\right\|_2^2 \\
&= \left\|e^{-\eta \boldsymbol{K} t}\boldsymbol{y}\right\|_2^2 \\
&= \left\|\boldsymbol{U}e^{-\eta \boldsymbol{\Sigma} t}\boldsymbol{U}^T\boldsymbol{y}\right\|_2^2 \\
&= \left\|\boldsymbol{U}\text{diag}([e^{-\eta\lambda_1 t}, \ldots, e^{-\eta\lambda_m t}]^T)\boldsymbol{U}^T\boldsymbol{y}\right\|_2^2,
\end{aligned}$$

where the last two equations follows the fact that $e^{-\eta \boldsymbol{K} t} = \boldsymbol{U}e^{-\eta \boldsymbol{\Sigma} t}\boldsymbol{U}^T$. This completes the derivation.

# D Experimental Details

## D.1 Training details

**Setup.** For all the tasks we train our model and the baselines using Adam [70] with maximum of 10,000 epochs and a learning rate of $3 \times 10^{-3}$. The batch size is 32 (*i.e.,* 32 rollouts), and we use a gradient clipping of 1.0 in terms of 2-norm to stabilize training. For the physics simulator, we use the four-step Runge-Kutta (RK4) as the numerical integration scheme in neural ODE. For computational resources, we train our model and the baselines in two machines: machine A has an Intel i9-9980HK CPU, and machine B has an Intel i7-6850K CPU with 4 NVIDIA GTX 1080 GPUs. In general, it takes about 8 hours to finish training. Fig. 6 plots the training curves of the four tasks.

**Instructions needed to reproduce the results.** The code can be found in the same zip file.

## D.2 Implementation and model architectures

We use the same architecture over our model in the visual tasks. A grid search is conducted to find the hyper-parameters of the model (*i.e.,* size of the networks, types of the activation function). We now describe each component in ALPS.

**The encoder.** To process pixel observations, the encoder uses a convolution neural network (CNN) with two convolutions and 2 by 2 max-poolings–the first convolution has a kernel size of 3 and a channel size of 3, and the second one has a kernel size of 4 and a channel size of 4, followed by vectorization to get a vector representation of an image. To provide the position information to this image representation, we obtain a positional embedding with the same size as the image representation using sine and cosine functions. We then concatenate these two vectors and feed them into a self-attention network, where the network has 10 attention heads and a relu activation function. Finally, the network outputs state estimations along the time horizon. Specifically, we only use the encoder part of the Transformer [54] without a dropout operation. We find that removing a dropout operation and using a relu activation function achieve the best result.

**The estimator.** For each element of the state predicted from the encoder, we construct a Fourier feature mapping. For example, the state of the pendulum consists of an angle and an angular velocity–we will have two vectors of the Fourier feature mapping here. Then, for each Fourier feature mapping, we use a residual network [1] with two residual blocks and a tanh activation function to get a representation. Finally, we concatenate these representations and feed them into an MLP to predict a system parameter. We find that using residual networks and a tanh activation function makes training faster and converges to a good result.

**The simulator.** The ODE solver takes in the predicted system parameter, and then rollouts state trajectories given the initial state predicted from the encoder.

**The decoder.** We use a two-layer MLP with a relu activation function and the size of 400, 400 to take in the simulated states and reconstruct the observations.

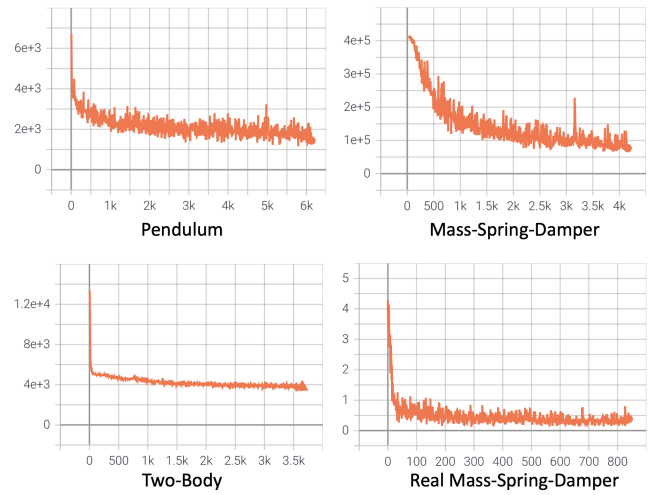



Figure 6: Learning curves during training of ALPS in the four tasks. Note that the weight

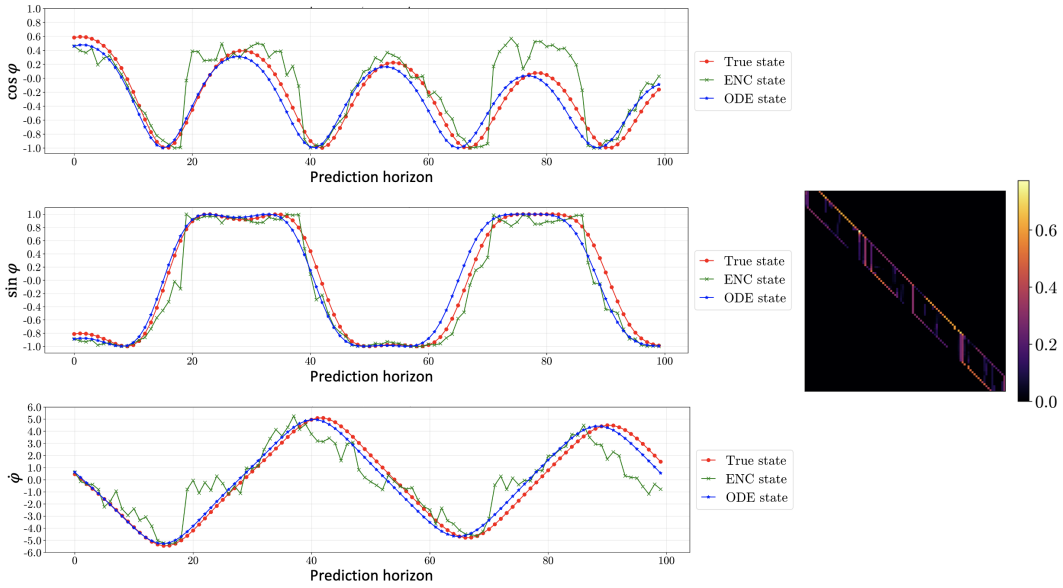



Figure 7: The state trajectories and the attention maps of the self-attention network in the pendulum task. We see that the self-attention network is able to infer states based on observations.

For the real MSD task, we remove the encoder and decoder in ALPS since we get noisy measurements of states in train wheel systems. For ALPS without Fourier features, we use the same architecture but the estimator becomes a single residual network that takes in an entire state trajectory in the time domain. For ALPS without self-attention networks, we use the same architecture but the encoder becomes a two-layer MLP with a relu activation that takes in image representations and predicts the position of an object. To estimate the velocity, we follow the procedure in [63], which uses a finite difference method.

Finally, for the baseline model CDM, we use the same model in [69], which consists of a context network (an MLP with the size of 400, 400), a forward network (an MLP with the size of 20, 20), and a backward network (an MLP with the size of 20, 20).

Note that CDM is trained with the loss function used in [69]; Autoencoder is trained with the VAE loss and observation reconstruction loss terms; the ablations of ALPS are trained with the same loss functions as the full model.

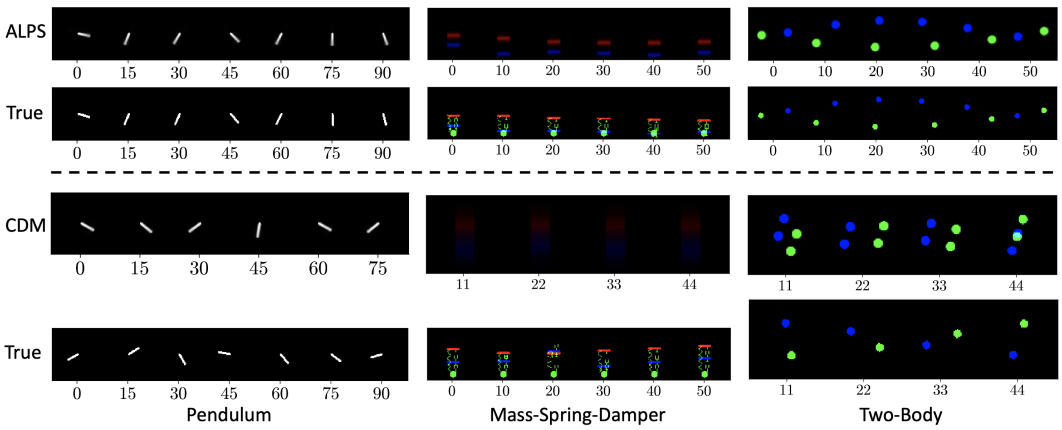



Figure 8: Examples of reconstructed and true observation sequences of ALPS and CDM over three visual tasks. We see that ALPS is able to robustly generate the observations.

# E Additional Results

## E.1 Visualization of encoded states prediction and attention map of the self-attention network

An important feature of ALPS is that it uses self-attention networks to infer states from pixel observations. Fig. 7 shows the encoded states $\{\tilde{\boldsymbol{x}}_s\}$, the ODE-simulated states $\{\hat{\boldsymbol{x}}_s\}$, and the true states $\{\boldsymbol{x}_s\}$ over time horizons $\tau$ as well as its attention map in the pendulum task. We see that the self-attention network is able to track the state evolution of the system from pixel observations. In addition, we observe that there is a small delay between the true states and the ODE-simulated states. This is due to a small prediction error in the initial state and the system parameter, which then accumulates over the time horizon.

For the attention map, the summation of each row is equal to one. This map shows that for each time step how much information of each observation in the entire trajectory is needed to predict the state at that time step. Here, we use a mask with the size of 15 to limit the attention width of the network. We see that the network mostly uses the observation in the beginning and the end of the attention mask to infer states. This suggests that the network learns to predict an *average* velocity. In addition, we see that there are vertical attention patterns, which happens to be at the peak of the cosine wave. We suspect that the network uses this information as an anchor to capture the periodic behavior of the pendulum. We leave this as a future work to understand the network.

## E.2 Visualization of reconstructed observations and prediction of system parameters

ALPS uses an autoencoder to learn system dynamics. Fig. 8 shows reconstructed observations of ALPS and the baseline in three visual tasks. We see that ALPS is able to accurately reconstruct the observations, whereas CDM fails to predict observations with duplicated particles in the Two-Body system and blur images in the MSD system. This implies that using physics in the loop can help to converge to a good solution.

In addition, Fig. 9 shows the system parameter prediction performance of ALPS over three visual tasks. Please see the caption for more details about the figure. We see that the model is able to reliably predict the system parameters in most cases. However, we also see there are some outliers for each group. We leave this as a future improvement.

## E.3 Analysis of self-attention networks

An important feature of ALPS is the use of self-attention networks for estimating states from observations. To provide the intuition about the mechanism of the self-attention network, we regress the self-attention network with true states of the system and visualize its attention weights on a simple

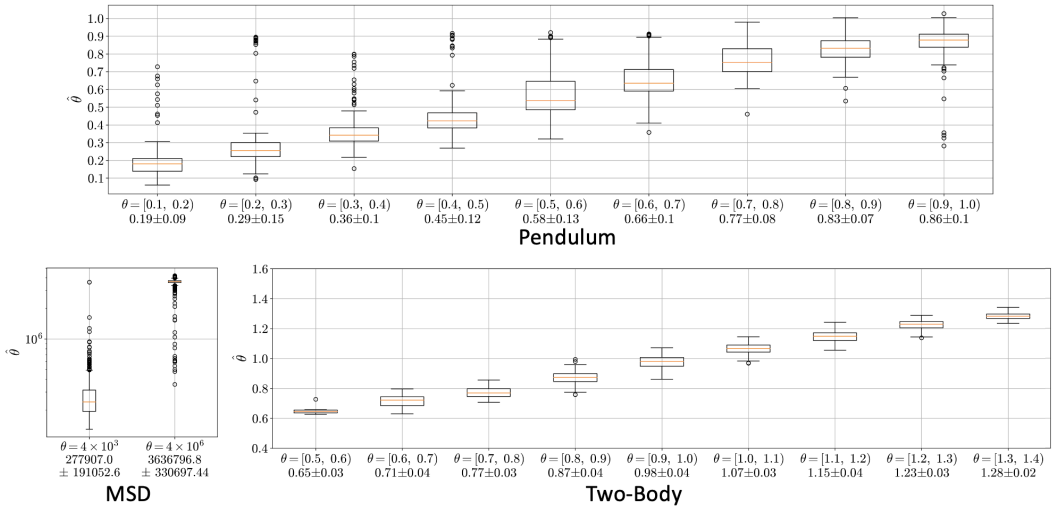
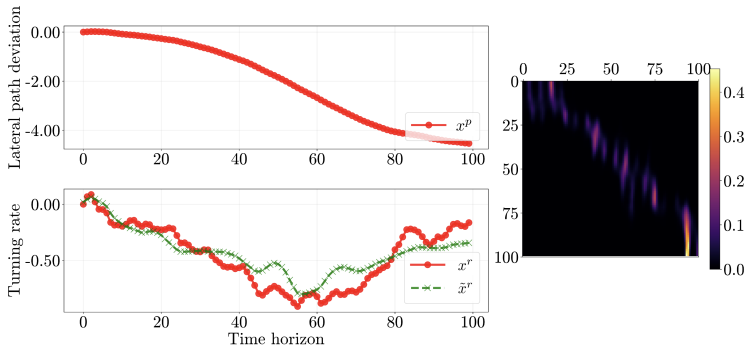



Figure 9: System parameter prediction performance of ALPS in the visual tasks. We categorize the value of the true system parameters, and show the box plot of the corresponding prediction values. The values at the bottom show the prediction mean and standard deviation of each group. We see that ALPS can identify the system parameters well.

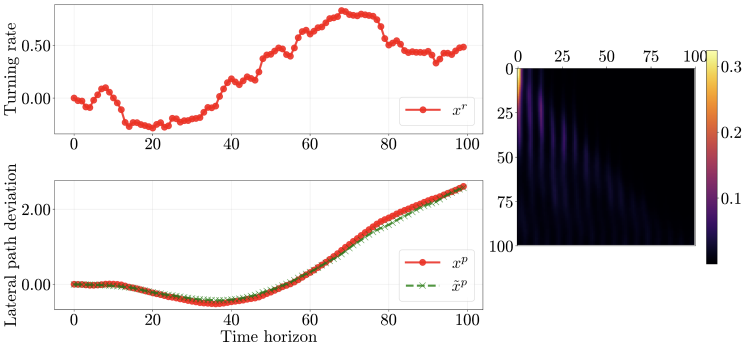
vehicle steering dynamics:

$$\frac{d\boldsymbol{x}}{dt} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 0.1 \\ 1 \end{bmatrix} u,$$

where the first element in $\boldsymbol{x} = [x^p, x^r]^T$ represents the lateral path deviation (*i.e.,* the distance between the vehicle and the center of the road), the second element represents turning rate, and $u$ is the input force to the steering wheel. In this example, we want to estimate either $x^p$ or $x^r$ by observing $x^r$ or $x^p$, respectively. Intuitively, in the first case (estimate $x^p$ from $x^r$), a self-attention network should perform *derivative* operation, whereas in the second case (estimate $x^r$ from $x^p$), a self-attention network should perform *integration* operation. Fig. 10 shows the result of the two cases. Here we use a single head self-attention network with a tanh activation function to regress the supervised data, and test on the testing data. In the first case the network attends to the neighboring observations at the current time (*i.e.,* diagonal pattern), whereas in the second case the network attends to the observations from beginning to the current time (*i.e.,* lower triangular pattern, global context). This observation suggests the network does exploit the local context to compute the gradient (*i.e.,* $x^r_t \approx \frac{x^p_{t+1} - x^p_t}{\Delta t}$), and use the global context to obtain the integral (*i.e.,* $x^p_t \approx \sum_{t'=0}^{t-1} x^r_{t'}$).




(a) Estimates the turning rate from the lateral path deviation




(b) Estimates the lateral path deviation from the turning rate

Figure 10: The patterns of the attention weight in the self-attention networks show the network attends to the local context (diagonal) to compute the gradient, and use the global context (lower triangular) to obtain the integral. **(a)** The case where the network estimates the turning rate $x^r$ from the lateral path deviation $x^p$ and its attention weight. **(b)** The case where the network estimates the lateral path deviation $x^p$ from the turning rate $x^r$ and its attention weight. The green lines $\tilde{x}^r$ and $\tilde{x}^p$ are the predictions and the red lines are the true values (turn the screen brighter to see the patterns).