EECS 16B Designing Information Devices and Systems II
Fall 2021

Note 9: Discretization and System ID

1 Controls Overview

Recall the big-picture motivations for the course all the way back in Note 0. We want to understand what it takes to create artificial systems that interact with the real dynamical world. One important component of this is being able to act in the real world to achieve our goals — without the ability to act in the world, it is hard to have an impact. Interacting with the real world to achieve our objectives is the subject of an area called sometimes called "control." Control is one of the foundational disciplines for the areas of robotics, artificial intelligence, and machine learning, while also playing very critical roles in networking, high performance computer systems, as well as power systems and a host of other areas within EECS and beyond.

Our entry into control will be through what is sometimes called "model-based control" — where we leverage explicit models for the world in order to plan and execute our interventions. How do we model the dynamic world? So far, we have seen the language of differential equations as a way to do this. We've done our examples for circuits — since they are the simplest aspect of the physical world to model in terms of dynamics — but the language of differential equations is also used to model mechanical systems, ecological systems, economic systems, financial systems, computational systems, etc. However, we face a significant challenge when it comes to actually making systems: the real world we want to impact exists in continuous time but the computational platforms that we need to use (i.e. digital computers and microcontrollers) cannot act infinitely fast and so, in order to be reliable while being flexible enough to be programmable, must act at discrete intervals.

Given a linear differential equation model for a system, we know how we can discretize it and obtain a discrete-time system model. You've already seen this in discussion and the homework, and we'll recap the story briefly in this note as well. However, this relies on knowing physical parameters to infinite precision and many times, it is not worth investing that much effort in trying to model the system exactly by hand. *System Identification* is about using data collected about the inputs $\vec{u}$ and states $\vec{x}$ to attempt to determine the parameters of the model. This process is of great importance, because when we construct real-world electromechanical systems, it is practically impossible to determine all their behaviors theoretically - we must do so empirically. In practice, data is also taken continuously to update our models, since in the real world, the model itself can change over time. The fact that all learning from data necessarily involves some residual inaccuracy is also going to be important in how we think about the use of such models.

Modern system design leverages the ability to learn from data. Fortunately, EECS16A has already given you all the basic tools that you need to do this intelligently, as we see in this note.

2 From continuous-time models to discrete-time ones

We have seen how to model a continuous-time system such as

$$\frac{\mathrm{d}}{\mathrm{d}t}\vec{x}_c(t) = A_c\vec{x}_c(t) + B_c\vec{u}_c(t). \tag{1}$$

What would be the counterpart for discrete-time? We could choose to sample our original signal and original input at a interval of Δ seconds and use those as our discrete state values to get

$$\vec{x}_d[i+1] = A_d\vec{x}_d[i] + B_d\vec{u}_d[i], \tag{2}$$

where $\vec{x}_d[i]$ is the value of $\vec{x}_c(i\Delta)$ and $\vec{u}_d[i]$ is the value of $\vec{u}_c(t)$ for the whole time interval $t \in (i\Delta, (i+1)\Delta]$. Note that if our input didn't actually have a constant input during the interval, then we are making an approximation of the true dynamics and so our discrete model won't be completely accurate to our continuous model.

Why would we need to do this? Because computers can only take in data in a sequence of snapshots taken with some spacing Δ between them (this is analogous to the clock rate of your CPU, but it is slower than that since you need to be able to do a nontrivial number of computations between samples in practice.) and for the same reason, can only change what computers put out into the world with some spacing Δ between outputs. Thus to be accurate to the real world, a computer needs a new *discrete-time* model of what is happening from one tick to the next Δ seconds later.

For the remainder of this module, we will almost entirely disregard the fact that our system may in fact really be continuous, focusing entirely on questions related to our discrete-time model. After all, in reality, most electronic control systems (like the MSP that we will use to develop a "robot car") have some intrinsic Δ in their ability to sample $\vec{x}_c(t)$ and vary their output $\vec{u}(t)$, so even if we knew that $\vec{x}_c(t)$ varied within the interval Δ, we would not be able to measure or react to any variation within the time interval.

2.1 Discretization for Scalar Differential Equations

In discussion, we talked about how to discretize a scalar differential equation given the assumption of piecewise constant inputs. This assumption is also referred to as a zero-order hold as the inputs can only be 0th order, i.e. constants, during each sampling interval. We originally did this so that we could take limits as $\Delta \to 0$, but it is useful in its own right given that computers deal with the world in discrete steps. We will redo the derivations here to refresh you and also generalize what we did to vector differential equations.

As a reminder from Note 1, for a scalar differential equation with a constant input

$$\frac{\mathrm{d}}{\mathrm{d}t}x(t) = \lambda x(t) + bu \tag{3}$$

where $\lambda \neq 0$ and initial condition $x(t_0)$ at time t_0, the solution is

$$x(t) = x(t_0)e^{\lambda(t-t_0)} + \frac{e^{\lambda(t-t_0)}-1}{\lambda}bu. \tag{4}$$

If $\lambda = 0$, then our differential equation simplifies to $\frac{\mathrm{d}}{\mathrm{d}t}x(t) = bu$ so our solution just becomes

$$x(t) = x(t_0) + bu(t-t_0). \tag{5}$$

When we want to discretize our continuous system from (1) into the form in (2), we need to calculate the state evolution from $\vec{x}_d[i] = \vec{x}_c(i\Delta)$ to $\vec{x}_d[i+1] = \vec{x}_c((i+1)\Delta)$.

For now assume our state is just a scalar. So during this sampling interval, our input is assumed to be constant at $u_d[i]$, which allows us to use the formula (4). Then we know that our initial condition is $x_d[i]$ at

time $t_0 = i\Delta$ and we want to evaluate our solution at time $t = (i+1)\Delta$. Thus assuming $\lambda \neq 0$, we get

$$x_d[i+1] = x_c((i+1)\Delta) \tag{6}$$

$$= x_c(i\Delta)e^{\lambda((i+1)\Delta - i\Delta)} + \frac{e^{\lambda((i+1)\Delta - i\Delta)} - 1}{\lambda} b u_d[i] \tag{7}$$

$$= x_d[i]e^{\lambda\Delta} + \frac{e^{\lambda\Delta} - 1}{\lambda} b u_d[i] \tag{8}$$

which matches the result from discussion. Finally if $\lambda = 0$, we get

$$x_d[i+1] = x_d[i] + \Delta b u_d[i]. \tag{9}$$

It is nice to see that $\lim_{\lambda \to 0} \frac{e^{\lambda\Delta}-1}{\lambda} = \Delta$ and so these two cases are actually consistent with each other. For this reason, we can just choose to write $\frac{e^{\lambda\Delta}-1}{\lambda}$ for all cases with the understanding that the expression can be unambiguously interpreted even at $\lambda = 0$.

2.2 Discretization for Vector Differential Equations

We now generalize what we've done to vector differential equations in order to fully discretize (1) into the form (2). Just like we did before with vector ODEs, we will assume that our transition matrix A_c has linearly independent eigenvectors and is thus diagonalizable so $A_c = V\Lambda V^{-1}$. Thus with a change of basis we can change variables to $\vec{z}(t) = V^{-1}\vec{x}_c(t)$. From Note 3, the system expressed in the eigenbasis coordinates then becomes

$$\frac{\mathrm{d}}{\mathrm{d}t}\vec{z}(t) = \Lambda\vec{z}(t) + V^{-1}B_c\vec{u}_c(t) \tag{10}$$

This system is now a set of independent scalar differential equations

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\vec{z}(t)\right)_k = \lambda_k\left(\vec{z}(t)\right)_k + \left(V^{-1}B_c\vec{u}_c(t)\right)_k \tag{11}$$

where for each equation we can get the corresponding solution from the previous section. Finally, we can stack the solutions into matrix-vector form and then convert back to the x-coordinates. For the A_d matrix, this means

$$A_d = V\begin{bmatrix} e^{\lambda_1\Delta} & 0 & \dots & 0 \\ 0 & e^{\lambda_2\Delta} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & e^{\lambda_n\Delta} \end{bmatrix} V^{-1}. \tag{12}$$

For the B_d matrix, we get

$$B_d = V\begin{bmatrix} \frac{e^{\lambda_1\Delta}-1}{\lambda_1} & 0 & \dots & 0 \\ 0 & \frac{e^{\lambda_2\Delta}-1}{\lambda_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{e^{\lambda_n\Delta}-1}{\lambda_n} \end{bmatrix} V^{-1}B_c \tag{13}$$

where as above, the expressions $\frac{e^{\lambda\Delta}-1}{\lambda}$ are to be interpreted as Δ when the $\lambda = 0$.

The cases where A_c is not diagonalizable can also be dealt with, but the key tool for doing so — upper

triangularization — will be coming later in this course.

2.3 Towards Euler Discretization

In practice, we like to discretize systems by using Δ as small as possible to have the most responsive system in terms of being able to control it as finely as possible. It is therefore interesting to see what happens in the limit of Δ being tiny.

By taking a Taylor expansion of the exponential, we see that $e^{\lambda\Delta} = 1 + \lambda\Delta + \frac{1}{2}(\lambda\Delta)^2 + \cdots$. What this means is that when we have a Δ that is much smaller than the reciprocal of the magnitude of the largest eigenvalue of A_c, then all these higher-order terms (quadratic or higher) in the Taylor expansion are tiny in comparison with the dominant terms. Remember, the error-term in the Taylor series for the exponential is quadratic after the linear term. And if $\lambda\Delta = 0.01$, then $(\lambda\Delta)^2 = 0.0001$ which is tiny when compared to $\lambda\Delta$ itself.

Let's use this to approximate A_d itself. In the following equations, we will use the little-o notation $o(x^2)$ which means any term that has the same or higher order of growth than x^2. This notation is out of scope of 16B but you will learn it in CS 61B and CS 170.

$$A_d = V \begin{bmatrix} e^{\lambda_1\Delta} & 0 & \dots & 0 \\ 0 & e^{\lambda_2\Delta} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & e^{\lambda_n\Delta} \end{bmatrix} V^{-1} \tag{14}$$

$$= V \begin{bmatrix} 1 + \lambda_1\Delta + o((\lambda_1\Delta)^2) & 0 & \dots & 0 \\ 0 & 1 + \lambda_2\Delta + o((\lambda_2\Delta)^2) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 + \lambda_n\Delta + o((\lambda_n\Delta)^2) \end{bmatrix} V^{-1} \tag{15}$$

$$= VIV^{-1} + \Delta V \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_n \end{bmatrix} V^{-1} + V \begin{bmatrix} o((\lambda_1\Delta)^2) & 0 & \dots & 0 \\ 0 & o((\lambda_2\Delta)^2) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & o((\lambda_n\Delta)^2) \end{bmatrix} V^{-1} \tag{16}$$

$$= I + \Delta A_c + V \begin{bmatrix} o((\lambda_1\Delta)^2) & 0 & \dots & 0 \\ 0 & o((\lambda_2\Delta)^2) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & o((\lambda_n\Delta)^2) \end{bmatrix} V^{-1} \tag{17}$$

$$\approx I + \Delta A_c \tag{18}$$

where the last approximation is just dropping terms that are all of order $(\lambda\Delta)^2$ or higher since they are very tiny when Δ is itself tiny.

What about B_d then? Here, there are more cancellations. Zooming in on the key term:

$$\frac{e^{\lambda\Delta} - 1}{\lambda} = \frac{1 + \lambda\Delta + o((\lambda\Delta)^2) - 1}{\lambda} \tag{19}$$

$$= \Delta + \frac{o((\lambda\Delta)^2)}{\lambda} \tag{20}$$

$$\approx \Delta \tag{21}$$

whenever Δ is tiny. This means that all of B_d can be approximated as ΔB_c.

It turns out that this approach can be generalized to approximate nonlinear differential equations in discrete-time whenever Δ is small. We will talk about this later in the course when we engage more significantly with nonlinear systems. This generalized approach is called *Euler discretization*.

3 System Identification

Note: Now that we are living in discrete time, we will drop the subscripts on x, A, B.

Instead of discretizing our continuous time model into a discrete one, what if we never had a precisely specified continuous time model to begin with? Is there some other way we can still determine the parameters of our discrete model relating $\vec{u}$ and $\vec{x}$?

Let our linear system be:

$$\vec{x}[t+1] = A\vec{x}[t] + B\vec{u}[t] + \vec{w}[t] \tag{22}$$

where we observe each of the $\vec{x}[t], \vec{u}[t]$ but A and B are unknown to us. Here, t refers to the current timestep and we include a disturbance term $\vec{w}[t]$ to capture external influences that nature might exert on our system. (Notice, this can in principle also include approximation errors that we might be making.) The important thing about the disturbance is that (a) we do not get to observe it directly; and (b) we hope that it is usually quite small. If it weren't usually small, we would be doomed.

From here on, it is understood that $\vec{x}[t]$ is discrete, so we will use subscripts to denote indices of the discrete-time vector (such as, $\vec{x}_1[t]$ is the value of the first component of $\vec{x}[t]$ at time t). Basically, there are going to be two different ways of indexing into these sequences of vectors. The $[t]$ will be used to index into a position in the sequence and the subscripts will be used to index into the vectors themselves.

Let's express our unknown matrices A and B in terms of scalars a_{ij}, b_{ij}, as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1k} \\ b_{21} & b_{22} & \dots & b_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nk} \end{bmatrix} \tag{23}$$

Substituting into our relation for $\vec{x}[t]$, we break down our state, observation, and input vectors into their scalar components as well:

$$\begin{bmatrix} x_1[t+1] \\ x_2[t+1] \\ \\ x_n[t+1] \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1[t] \\ x_2[t] \\ \vdots \\ x_n[t] \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1k} \\ b_{21} & b_{22} & \dots & b_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nk} \end{bmatrix} \begin{bmatrix} u_1[t] \\ u_2[t] \\ \vdots \\ u_k[t] \end{bmatrix} + \vec{w}[t] \tag{24}$$

Now comes the interesting part. Recall that our unknowns are in fact all the a_{ij} and b_{ij} scalars, and our knowns are all the components of $\vec{x}$ and $\vec{u}$ (x_1, x_2, u_1, u_2, etc.) When solving linear systems, we know that

we should put our unknowns into a vector. By considering each row of the above matrix separately (using r to index the row), we obtain scalar equations of the form:

$$x_r[t+1] = \begin{bmatrix} a_{r1} & a_{r2} & \cdots & a_{rn} \end{bmatrix} \begin{bmatrix} x_1[t] \\ x_2[t] \\ \vdots \\ x_n[t] \end{bmatrix} + \begin{bmatrix} b_{r1} & b_{r2} & \cdots & b_{rk} \end{bmatrix} \begin{bmatrix} u_1[t] \\ u_2[t] \\ \vdots \\ u_k[t] \end{bmatrix} + w_r[t] \tag{25}$$

for all $1 \leq r \leq n$. Notice that the two terms on the right of the equation are really just inner products, producing a scalar. As the inner product is symmetric for real vectors $\big(\langle x, y \rangle = \langle y, x \rangle\big)$, we can swap the rows and columns and rewrite the above equation as:

$$x_r[t+1] = \begin{bmatrix} x_1[t] & x_2[t] & \cdots & x_n[t] \end{bmatrix} \begin{bmatrix} a_{r1} \\ a_{r2} \\ \vdots \\ a_{rn} \end{bmatrix} + \begin{bmatrix} u_1[t] & u_2[t] & \cdots & u_k[t] \end{bmatrix} \begin{bmatrix} b_{r1} \\ b_{r2} \\ \vdots \\ b_{rk} \end{bmatrix} + w_t[t] \tag{26}$$

again for all $1 \leq r \leq n$.

Combining the two terms on the right, we finally obtain the family of equations

$$x_r[t+1] = \begin{bmatrix} x_1[t] & x_2[t] & \cdots & x_n[t] & u_1[t] & u_2[t] & \cdots & u_k[t] \end{bmatrix} \begin{bmatrix} a_{r1} \\ a_{r2} \\ \vdots \\ a_{rn} \\ b_{r1} \\ b_{r2} \\ \vdots \\ b_{rk} \end{bmatrix} + w_r[t] \tag{27}$$

again for all $1 \leq r \leq n$.

Notice that we have managed to place all the unknowns involving the rth element of the state at time t in a single vector. Let's give it the name $\vec{p}_r = \begin{bmatrix} a_{r1} \\ a_{r2} \\ \vdots \\ a_{rn} \\ b_{r1} \\ b_{r2} \\ \vdots \\ b_{rk} \end{bmatrix}$ since it represents the r-th row of unknown **p**arameters.

Since we now have a linear equation with our unknowns in a vector, are we done, since we can do Gaussian elimination to solve for the unknowns?

Unfortunately not. For one, we have these unknown small disturbances $w_r[t]$ corrupting our equations. Second, we have $n + k$ unknowns, but only a single equation so far. We need more equations.

Recall that our model eq. (22) is presumed to hold for *all* values of t, since we are assuming that our system's behavior does not vary over time. Notice that for all timesteps $0 \leq t \leq T$, where T is the current time, the vector of unknowns remains the same. Thus, by considering all these values of t and stacking them vertically, we obtain the matrix system of equations

$$\vec{s}_r = \begin{bmatrix} x_r[1] \\ x_r[2] \\ \vdots \\ x_r[T] \end{bmatrix} = \begin{bmatrix} x_1[0] & \cdots & x_n[0] & u_1[0] & \cdots & u_k[0] \\ x_1[1] & \cdots & x_n[1] & u_1[1] & \cdots & u_k[1] \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ x_1[T-1] & \cdots & x_n[T-1] & u_1[T-1] & \cdots & u_k[T-1] \end{bmatrix} \vec{p}_r + \begin{bmatrix} w_r[0] \\ w_r[1] \\ \vdots \\ w_r[T-1] \end{bmatrix} \tag{28}$$

which consists of T equations, not just 1! For $T \geq n+k$, we can use least squares to estimate our unknowns $\vec{p}_r$ if we count on the hope that all the $w_r[t]$ are small on average. Here, we introduced some notation $\vec{s}_r$ for the vector that consists of the trace of the r-th state. We can also define the data matrix

$$D = \begin{bmatrix} x_1[0] & \cdots & x_n[0] & u_1[0] & \cdots & u_k[0] \\ x_1[1] & \cdots & x_n[1] & u_1[1] & \cdots & u_k[1] \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ x_1[T-1] & \cdots & x_n[T-1] & u_1[T-1] & \cdots & u_k[T-1] \end{bmatrix}. \tag{29}$$

With this notation in hand, what is this least-squares estimate for our parameters $\vec{p}_r$ from the r-th row? From 16A, we remember the formula:

$$\widehat{\vec{p}}_r = \left(D^\top D\right)^{-1} D^\top \vec{s}_r. \tag{30}$$

Here, we use the "hat" notation $\widehat{\vec{p}}_r$ to remind ourselves that this is an estimate of the parameters, not the true parameters themselves. After all, least-squares typically has some error.

We can repeat evaluations of the formula in eq. (30) for each value of r in order to fully determine our system's parameters. However, it is interesting to notice something. The matrix $\left(D^\top D\right)^{-1} D^\top$ has no dependence on the specific row r — it is the same for all the rows. So we don't have to recompute this for each r.

Is there a compact way of representing the same matrix multiplying many different vectors? We know from 16A that indeed there is, this is the very definition of matrix matrix multiplication from a column perspective. This means that we can define a pair of new matrices by arranging $\vec{p}_r$ together into a horizontal stack, as well as $\vec{s}_r$.

We can create a matrix P of all our unknowns across all values of r by stacking them horizontally to obtain

$$P = [\vec{p}_1 \vec{p}_2 \cdots \vec{p}_n] = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{12} & a_{22} & \dots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{nn} \\ b_{11} & b_{21} & \dots & b_{n1} \\ b_{12} & b_{22} & \dots & b_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ b_{1k} & b_{2k} & \dots & b_{nk} \end{bmatrix} = \begin{bmatrix} A^\top \\ B^\top \end{bmatrix}. \tag{31}$$

We can follow a similar stacking procedure to obtain

$$S = [\vec{s}_1 \vec{s}_2 \cdots \vec{s}_n] = \begin{bmatrix} x_1[1] & x_2[1] & \cdots & x_n[1] \\ x_1[2] & x_2[2] & \cdots & x_n[2] \\ \vdots & \vdots & \ddots & \vdots \\ x_1[T] & x_2[T] & \cdots & x_n[T] \end{bmatrix} = \begin{bmatrix} \vec{x}[1]^\top \\ \vec{x}[2]^\top \\ \vdots \\ \vec{x}[T]^\top \end{bmatrix}. \tag{32}$$

With this notation in hand, we can compactly express the repeated application of eq. (30) as

$$\widehat{P} = \left(D^\top D\right)^{-1} D^\top S. \tag{33}$$

In fact, Python packages support the solving of matrix approximate equations of this form:

$$DP \approx S \tag{34}$$

using least-squares.

It is good to step back and ask why do we use least-squares here? Because we are going to assume that any measurement errors or unmodeled terms are all relatively small. So it makes sense to pick a solution that has a small residual, and least squares gives that to us. (We can see in another note how we could make such solutions robust to a few points being hit with some much larger measurement noise or just unmodeled behavior.)

Recall from 16A that $D^\top D$ is invertible exactly when D has linearly independent columns. We will discuss what happens when the columns made out of the x_r are dependent later — for now, we will simply comment that choosing our control inputs randomly will ensure with high probability that the input columns are all linearly independent both of each other and of the earlier x_r columns.

4 Final Comments

The above recipe is pretty generic. We write out our model and figure out what are parameters that we need to learn. Then, we use the data that we have collected to set up systems of approximate equations in the unknown parameters that we need to learn. Finally, we use least-squares to solve for those parameters. If our measurements are of high quality and our model is a reasonable match to reality, then we will get estimated values for the parameters that work well.

There remains only one more question. How can we tell if the estimate that we compute for the parameters P is any good? In all learning-based approaches, the ultimate evaluation of learned parameters comes from actually using them to achieve whatever goal it is that we want to achieve. For systems that are to be controlled, that goal is typically the successful control of such systems. If the system can be successfully controlled in reality, then the system identification was clearly good enough.

However, in many learning contexts, we do not have access to the complete engineered system yet. Seeing how the final system works is the engineering counterpart of a functional test in software engineering. When the control performance is itself only a part of a larger system, then testing control performance using the identified parameters is the counterpart of an integration test in software engineering. This prompts the natural question: what is the learning-engineer's counterpart of a unit test in software engineering? Is there any way that we can see if the learned parameters are credibly correct without having to integrate with control?

The answer to this question is yes, as you saw in 16A. The key is to test the prediction performance of the model using some extra data that we did not use to learn the parameters. After all, if we were actually able to learn something close to the true performance, we should be able to predict the next state from the previous state and the control input.

It is easiest to understand this in the scalar context from a notational perspective, but the same idea applies in the vector case. Suppose that we have another trace of states and inputs: $\check{x}[0], \check{x}[1], \ldots, \check{x}[T]$ and $\check{u}[0], \check{u}[1], \ldots, \check{u}[T-1]$. This trace was not used to learn the estimated parameters $\widehat{a}, \widehat{b}$ for the scalar system. We can evaluate the prediction error for each state by computing $(\check{x}[1] - \widehat{a}\check{x}[0] - \widehat{b}\check{u}[0]), (\check{x}[2] - \widehat{a}\check{x}[1] - \widehat{b}\check{u}[1]), \ldots, (\check{x}[T] - \widehat{a}\check{x}[T-1] - \widehat{b}\check{u}[T-1])$ and can then compute the average squared prediction error using $\frac{1}{T}\sum_{k=1}^{T}(\check{x}[k] - \widehat{a}\check{x}[k-1] - \widehat{b}\check{u}[k-1])^2$. If this is close to or below the training error $\frac{1}{T}\sum_{k=1}^{T}(x[k] - \widehat{a}x[k-1] - \widehat{b}u[k-1])^2$ on the data used to learn the model, then it reasonable to have some confidence in the validity of the learned model parameters — especially if this average error is low.

Here, it is ideal for the "test data" to be drawn from a different trace from the same physical system since presumably, the eventual goal is to control that same physical system over different runs. The use of test data from the same trace is also possible if the corresponding equations had not been used while generating the D matrix for training, however the use of such tests gives less confidence that the learned parameters will perform successfully in a new run of that physical system.

This approach of keeping some collected data aside to test the quality of learning is something that is quite generally practiced. In 16B, you see machine learning properly situated within a larger engineering context. In other courses on machine learning (like 189), these contexts are often missing and so "performance on test data" can seem to be a larger goal than the unit test that it actually is.

Contributors:

- Neelesh Ramachandran.
- Rahul Arya.
- Druv Pai.
- Anant Sahai.
- Ashwin Vangipuram.

